2004-04-01

Source Level Debugging of Circuits Synthesized from High Level Language Descriptions

Karl S. Hemmert
Brigham Young University - Provo

SOURCE LEVEL DEBUGGING OF CIRCUITS SYNTHESIZED FROM HIGH LEVEL LANGUAGE DESCRIPTIONS

by

Karl S. Hemmert

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

Brigham Young University

August 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Karl S. Hemmert

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Brad L. Hutchings, Chair
Date	James K. Archibald
Date	Brent E. Nelson
Date	Doran K. Wilde
Date	Michael J. Wirthlin

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Karl S. Hemmert in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Brad L. Hutchings
Chair, Graduate Committee

Accepted for the Department

Michael A. Jensen
Graduate Coordinator

Accepted for the College

Douglas M. Chabries
Dean, College of Engineering and Technology

ABSTRACT

SOURCE LEVEL DEBUGGING OF CIRCUITS SYNTHESIZED FROM HIGH LEVEL LANGUAGE DESCRIPTIONS

Karl S. Hemmert

Electrical and Computer Engineering

Doctor of Philosophy

The rapid increase in the density of modern FPGAs has allowed ever increasingly complex designs to be mapped to FPGAs. However, this increase in logic resources is accompanied by an increase in the complexity of describing and verifying the operation of an application. This has prompted the search for new approaches to the design, debug and verification of circuits. The desire to find more effecient approaches to designing these large FPGA circuits has led to the creation of synthesizing compilers that can create hardware from high-level descriptions based on general purpose programming languages. Being able to describe the application at a high level of abstraction allows the designer to focus on the algorithms, rather than the implementation details. Though synthesizing compilers can make it easier to create circuits, they can make it more difficult to debug the resulting circuit. Typically, a design is debugged and verified by simulating the application/circuit in software (possibly at many different levels of abstraction). However, because of the reprogrammability of FPGAs, it is possible to use the FPGA device directly during the debug process. Performing design debug verification in the FPGA device has two significant advantages. First, the debugging occurs in the hardware itself and not a

virtual abstraction. Second, debugging in hardware occurs at hardware speeds, which is orders of magnitude faster than software simulation. These two advantages make it possible to continue to verify large FPGA based designs.

ACKNOWLEDGMENTS

There are a number of people I would like to thank for their contributions and support to this project and to my education. First, I would like to thank my advisor, Brad Hutchings, for giving a physicist turned electrical engineer an opportunity to pursue a PhD in the first place. The research position he afforded me was a valuable opportunity for learning and growth. I would also like to thank the other members of my committee, James Archibald, Brent Nelson, Doran Wilde, and Michael Wirthlin, for their support and influence throughout my academic career.

I would be remiss if I didn't thank Justin Tripp and Preston Jackson for their patience as I asked endless questions and modified their code to meet my needs. Without their work, this dissertation would not have been possible. I would also like to thank the other members of the Configurable Computing Lab, past and present, for their friendship and help. I regret that there are too many to name individually (and I would feel badly if I left one out).

Most importantly, I would like to thank my wife Lindsey for being so patient for all these long years. Without her support I would have never made it through. I am also grateful for my two boys, Joshua and Aaron, who provide such joy and much needed stress relief. I am also grateful to my parents, Kelly and Darla Hemmert, for their patience and their gentle prodding. Nothing of this magnitude can be done without the support of family and loved ones.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In 1965, Gordon Moore [1] made an observation about the exponential increase in the number of transistors in an integrated circuit. This observation later became known as Moore's Law, and in its current form states that the number of transistors on a chip doubles every 18 months. This prediction of frequent doubling in circuit density has held true for four decades, and it appears that it will continue for at least another decade, leading to the production of ever larger circuits.

While enabling a revolution in computing, this exponential increase in circuit size has caused other interesting problems. The main problems are found in the design, verification and debug of such large circuits. On the design side, this has led to larger design teams and the use of automated Computer Aided Design (CAD) software to more quickly implement circuitry.

The verification and debug of these larger circuits is even more troublesome because the possible states in a design goes up exponentially with the size of the circuit. Since the potential size of these designs is also going up exponentially, it is impossible to fully verify the state space of modern designs. Because of this, a design must be selectively verified, using targeted and pseudo-random test vectors [2]. However, the number of these selective tests is also increasing rapidly. For example, Sun Microsystems [3] reports that in 1991 the typical design required 4 million lines of verification code. By 2000, this number had increased to over 400 million, a one hundred times increase in only 9 years.

This increase in validation complexity is further complicated by the need to use simulation for much of the verification process. The use of simulation is necessitated by the large investment in time and money required to fabricate a design, which is a prerequisite

to verifying the design in hardware. This leads to the general design cycle shown in Figure 1.1, which will be referred to as the ASIC (Application Specific Integrated Circuit) Design Cycle. The impact of this can be seen in Sun Microsystems' report that the 400 million lines of validation code required over 35 CPU years to execute.

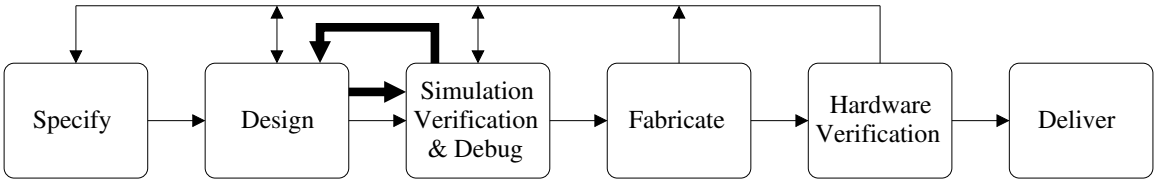


Figure 1.1: ASIC Design Cycle. The thick lines indicate the typical inner loop of the design cycle.

There are two types of costs when fabricating an ASIC. The first is the monetary cost, which can be upward of one million dollars to produce a prototype chip. The second (and more significant as it applies to debugging) cost is the time it takes to get hardware back from the fabrication process; this time can be as long as several months. This creates a large cost of failure when fabricating a design and leads to a large amount of front-end simulation. For this reason, most of the time in the ASIC Design Cycle is spent in simulation and debug (see the bold cycle shown in Figure 1.1). This is problematic because the number of computations required to simulate a circuit is increasing with the size of circuits; as circuit size doubles, the number of computations to simulate a single clock cycle of circuit operation also approximately doubles.

The increase in the number of verification tests and the increase in the number of computations required to simulate increasingly larger circuits leads to about a four times increase in computational requirements to debug a design for each doubling in circuit size. Because the performance of microprocessors is only nearly doubling every eighteen months, this is an important issue; it means that in any given eighteen month period, computational complexity of verifying a circuit quadruples, but microprocessor performance only

doubles. For this reason, the inability to debug the final hardware is becoming more and more of a problem in the ASIC community. However, there are families of programmable devices, known as Field Programmable Gate Arrays (FPGAs), for which these limitations do not apply. The next section will look at how these devices are different and how these differences can be exploited.

1.1 Debugging FPGA-Based Applications

Moore's Law is having the same effect on FPGAs as it is on ASICs; every 18 months the available resources in an FPGA approximately double. Just as with ASICs, increased design complexity has led to a search for more efficient ways to describe circuits at a high level of abstraction. Several synthesizing compilers that generate circuits from a description written in a high level language [4, 5, 6, 7, 8, 9, 10] have been created. Many of these languages are based on a general purpose programming language. However, while these synthesis tools help FPGA designers to manage the complexity of their designs, these tools usually ignore an important advantage that the FPGA has over the ASIC: its immediate availability.

This thesis is based on the following premise: FPGA applications should be debugged in FPGA *devices*. FPGA devices are typically available at the beginning of a design cycle and it makes good sense to use them during functional verification, just as software is verified using microprocessor devices – not microprocessor simulations.

Because FPGA hardware can be immediately available to the designer, it should be possible to functionally verify and debug a design directly in the device. This provides two advantages. First, the debug and verification of the design can take place in the hardware itself and not a virtual abstraction of that hardware, such as a logical or gate level simulator. This guarantees that a software abstraction of the design is not debugged only to find that the actual circuit does not function properly. Second, debug and verification in hardware occurs at hardware speeds, which can be orders of magnitude faster than software execution. In contrast to ASIC design, these two advantages make it possible to continue to effectively and quickly debug and verify large FPGA based designs, even with the continual increase of design size. This leads to a slightly different inner loop for the design cycle

as compared to ASICs. This is shown in Figure 1.2. The ability to move directly to hardware allows the designer to bypass the slow simulation phase and go straight to hardware[1]. This can greatly increase the productivity of the designer, especially when debugging large circuits.

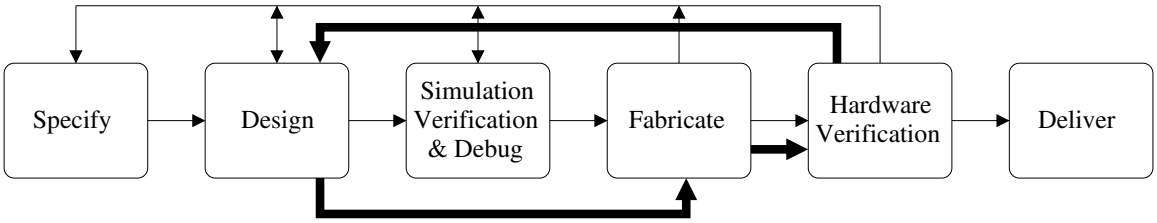


Figure 1.2: FPGA Design Cycle. The thick lines indicate the inner loop of the design cycle.

Despite these advantages, functional design verification and debug for FPGA-based applications is typically performed in software simulation, similar to the ASIC Design Cycle discussed above. This is primarily due to a lack of debugging tools that: 1) provide the controllability and observability necessary to functionally verify operating hardware, and 2) allow the designers to debug their applications using the same abstractions that were used to design them.

This dissertation demonstrates that a direct debugging tool with sufficient controllability and observability can be a powerful hardware debugging aid. Providing control and feedback to the user at the design level of abstraction provides the user a familiar view of the application while debugging. This dissertation concentrates on the debug of circuits described in a high level language and synthesized to hardware using a synthesizing compiler.

To demonstrate the effectiveness of debugging FPGA applications directly in the FPGA device, we have created a source level debugger for the Sea Cucumber (SC) synthesizing compiler [10]. The SC Debugger [11, 12] distinguishes itself from the source

[1]Of course simulation is still available for use, if desired. This can be useful for verifying small parts of the circuit.

level debugging tools which come with typical synthesizing compilers, such as those which support SystemC [6] and Handel-C [8], by the fact that it is used to debug the final circuit, rather than the software description. Currently available debuggers allow the user to debug the *software execution* of the application at the source level; the SC Debugger provides source level debugging of actual *circuit operation*, by correlating the state of the executing circuit with the original source code. This allows the user to take advantage of the speed of hardware execution, while still allowing debug information to be displayed in the context of the original source code.

1.1.1 Debugger Features

Because this work deals with circuit descriptions made in languages which resemble general purpose programming languages, the feature set of the hardware debugger was chosen to closely parallel the controllability and observability of a typical software debugger. The feature set of the hardware debugger includes the following:

1. *Single-stepping.* Ability to single-step the execution of the program.
2. *Breakpointing.* Ability to allow execution to run to arbitrary points in the code.
3. *Setting values of variables.* Ability to set the values of variables found in the source code.
4. *Location of current execution points.* Show the current execution points while execution is paused.
5. *Watching values of variables.* Ability to watch the value of variables in the program.

Although the SC debugger is modeled after software debuggers, there are some fundamental differences between software and hardware debug. This is exemplified by the two debugging approaches provided by the SC debugger: software-centric and hardware-centric. The two approaches are distinguished by how the single step command operates. In the hardware-centric view, the executing circuit is viewed as a parallel machine, and a single step is defined as allowing the circuit clock to advance one clock cycle. In this

approach, the debugger attempts to show the user exactly what is happening to the circuit state. In the software-centric approach, the debugger creates the illusion that the hardware is executing sequentially. This is done by defining a single step as advancing the execution one line of source code, just as in a software debugger. In this mode of operation, the debugger uses Virtual Sequentialization (discussed in Chapter 9) in order to make the parallel hardware appear to execute sequentially. The SC Debugger provides both of these modes of operation. The hardware-centric approach is called clock step mode and the software-centric view is called source step mode. These two modes will be discussed in Chapters 8 and 9, respectively.

In addition to providing the above capabilities, the debugger work concentrates on supporting debugging in the presence of optimizations which are typically used when mapping code to Very Long Instruction Word (VLIW) machines that use predication (which are also used by Sea Cucumber to implement circuits). The main optimizations considered in the work are predication, static single assignment (SSA), block merging and instruction scheduling. Other types of optimizations will also be briefly discussed, however, this work will not address automated loop unrolling or pipelining.

1.1.2 Platform Requirements

To enable hardware debugging, the target platform must provide a minimum amount of controllability and observability. This includes the ability to control the circuit clock and read and write circuit state. If these features are not provided by the vendor of the platform, the synthesizer can take advantage of the reconfigurability of FPGAs to add circuitry to provide the features. However, it is always preferable to have the features supported directly by the platform. Chapter 7 will discuss this subject in some detail.

1.1.3 Debug Database

As with software debuggers, the SC Debugger needs information provided by the compiler in order to function. We refer to this information as the debug database. As part of this project, SC was modified to provide this information with the output of compilation. The debug database provides the debugger with complete two-way mappings to

associate the elements in the source code with the circuitry produced through the synthesis process. These mappings are similar to, but much more complex than, those provided by software compilers in the symbol table, and includes information about how control- and data-flow are mapped in the final application. The contents of the debug database, along with the issues involved in its creation is discussed in Chapter 6.

1.2 Contributions

The Sea Cucumber Debugger effort has resulted in the following contributions. These contributions have been incorporated into the synthesis tool and debugger which provide a complete two-way mapping between source code and hardware.

1. Information about and control of the executing circuit is provided to the user in the context of the original source code.

2. The feature set of the hardware debugger is similar to that of a software debugger, including single-stepping, breakpointing, and setting and watching of variable values.

3. The debugger allows full visibility into the operation of the synthesized circuit, providing information at both the source and circuit levels.

4. The debugger is capable of operating in the presence of predication, static single assignment, block merging and instruction scheduling.

5. The debugger is capable of working both with a simulation of the circuit and during actual execution on the FPGA.

1.3 Outline of Dissertation

The remainder of this work is dedicated to a detailed discussion of the different parts needed to create a source level debugger for synthesized circuits. Chapter 2 will first discuss related work in both the area of software debuggers and hardware debugging. Chapters 3, 4 and 5 provide background necessary for a better understanding of the current work: Chapter 3 provides general information about the main optimizations considered in this work; Chapter 4 provides information about the features required to provide the

controllability and observability necessary to enable hardware debug of an FPGA-based platform; and Chapter 5 gives a brief description of how the Sea Cucumber Synthesizing Compiler operates.

After the background information is provided, the remaining chapters are dedicated to describing the novel work conducted for this dissertation. Chapter 6 discusses the issues involved with creating the debug database. Chapter 7 discusses hardware which is added by the synthesizer to help enable the debugger feature set. Chapters 8 and 9 discuss, in general terms, the issues in implementing clock step and source step modes, respectively. Chapter 10 then discusses the actual implementation of the SC Debugger. Conclusions and suggestions for future work are found in Chapter 11.

Chapter 2

Related Work

This work combines ideas from two distinct areas of research: source level debugging of optimized programs and debugging of FPGA-based applications. The first area includes research into how to debug programs that have been optimized during compilation, and are targeting general purpose processors. The second area looks at debugging FPGA-based applications at the same level of abstraction at which they are designed.

This chapter will discuss the research and products that have had the greatest impact on the current work on the SC Debugger. In order to more clearly describe the work, this chapter is broken into two sections, one for each area of research.

2.1 Source Level Debugging of Optimized Programs

Because of the number of optimizations which take place during the synthesis process, the debugging of circuits synthesized from high level design languages has some similarities with the debugging of optimized code in CPUs. Understanding the issues involved in source level debugging of optimized programs helps clarify some of the issues involved in debugging synthesized circuits at the source level. However, these ideas have limited direct application because the results of software compilation and hardware synthesis are vastly different.

The following are short descriptions of some previous work done to facilitate the debugging of optimized programs. Note that while many of these approaches have utility, none of the approaches described below have made it into any mainstream debuggers of which the author is aware.

2.1.1 Interactive Source-Level Debugging of Optimized Programs [13] by Polle Zellweger

This is one of the earliest works looking at source level debugging of optimized code. It deals with a limited number of optimizations, but an actual compiler and debugger were built that could handle those optimizations. The debugger provided *expected* behavior where possible and *truthful* behavior at other times. It was considered better to provide the behavior which is expected when debugging unoptimized code and resorting to truthful behavior only when necessary.

This work also introduced terms to describe discrepancies that occur when defining breakpoints for optimized code. In unoptimized code, a breakpoint can be set *between* lines of code. When such a breakpoint becomes active, the user can assume that all instructions prior to the breakpoint have executed and that no instructions past the breakpoint have executed. In optimized code, such locations are not guaranteed to exist. Zellweger used the term "semantic" breakpoint to refer to a breakpoint which is set before the first instruction on the desired line. A "syntactic" breakpoint is set after the last instruction originating from the previous line. When debugging unoptimized code, these two breakpoints converge to the same point.

2.1.2 Debugging Parallelized Code Using Code Liberation Techniques [14] by Patricia Pineo and Mary Soffa

This work provides insight into the debugging of parallelized code. The target for the compiler is a multi-processor system. Though many people advocate debugging such code in a uniprocessor system, with the code running sequentially, the authors advocate debugging the parallelized code running on a multi-processor system. However, they feel that the debugging should be done in the context of the original sequential code, a sentiment shared in the goals of the SC Debugger work. The main technique used during compilation is single assignment. This means that each variable is assigned only once. Under single assignment, scalar values used in loops become arrays, and arrays in loops become arrays of arrays. This technique allows the compiler more freedom in applying parallelizing transformations to the code by removing all pseudo-dependencies from the data-flow.

Name reclamation, a technique where the compiler attempts to minimize memory requirements of a program through memory reuse, is the most important contribution of this work in the context of hardware debugging. The compiler performs name reclamation by finding versions of variables which do not have overlapping regions of validity in the program. These variables are allocated to shared memory locations. However, the compiler will not allocate variables to shared memory locations if doing so will make a variable value unavailable at runtime for debugging purposes. This extends the variable lifetime longer than is strictly necessary to ensure that the value is available to the debugger. The one exception to this is if the statement that assigned the variable is moved ahead of the current breakpoint during optimizations. This is a restriction shared by other projects [15, 13] which deal with optimized code.

The SC Debugger discussed in this dissertation also uses a buffering technique to allow the current values of variables to be retrieved at runtime. However, the SC Debugger is an improvement in that the approach taken allows the debugger to determine the values of variables whose assignments have been moved beyond the currently executing line. This is discussed in Chapter 9.

2.1.3 Debugging VLIW Code After Instruction Scheduling [16] by Lyle Cool

This masters thesis does a good job of summarizing some of the major work done in the field of source level debugging of optimized programs as well as presenting motivation for debugging the optimized version of the program instead of debugging the unoptimized form. Cool argues that some bugs only manifest themselves when optimizations are turned on. This may arise because of bugs in the compiler or because of "unsafe" optimizations which the compiler may perform.

Cool builds on the work done by Zellweger to enable the debugging of VLIW code after instruction scheduling has taken place. However, because he is working with VLIW hardware, he has to deal with multiple instructions being active at one time, as well as the possibility that instructions have been re-ordered in order to take advantage of

instruction level parallelism (ILP). Similar problems are seen when debugging synthesized hardware at the source level.

Cool proposes changes to the symbol table and other debug information to handle code duplication, code reordering, and code compaction. Because he is working with VLIW code which will execute multiple instructions at the same time, his example implementation uses code highlighting to specify which instructions are in operation. He also proposes that when necessary the user should be shown the results of the optimizations in order to clarify what is actually happening in execution. The SC Debugger work uses a similar mechanism to show executing lines of code when showing the user *truthful* behavior in the hardware-centric approach (see Chapter 8).

2.1.4 Debugging Optimized Code Without Being Misled [17] by Max Copperman

Copperman describes a graph based approach to solving the problems associated with debugging optimized programs. His approach requires that the debugger have access to the unoptimized program, as well as the optimized program, both created by the same compiler. The paper concentrates on determining the currency of variables so that the user is not misled by false variable values. This is accomplished by comparing the control- and data-flow graphs of the optimized and unoptimized programs. This method can be applied to a wide range of optimizations. The one restriction is that the control-flow of the optimized program cannot be too different from the control-flow of the unoptimized program. This constraint renders this technique infeasible for debugging synthesized circuits because synthesizers typically employ optimization techniques that significantly affect control flow. However, the SC Debugger work does utilize the idea of keeping a copy of the original, unoptimized control-flow and data-flow of the program to help enable certain debugger features (see Chapter 9).

2.1.5 A New Framework for Debugging Globally Optimized Code [18, 19] by Le-Chun Wu

This work uses the IMPACT [20] compiler to create debug information that supports source level debugging of optimized code. The debugger developed for this project

recovers and displays *expected* behavior, whenever possible, and addresses both code location as well as data value problems. It does this by using a novel forward recovery algorithm. The algorithm works as follows: A statement breakpoint stops program execution before any successive (past the breakpoint) operation is executed. The point at which the execution halts is called the *interception point*. After the execution of the application is stopped, the debugger continues by partially emulating all instructions expected to execute before the breakpoint. At this point, the debugger responds to user actions based on the information in the actual program execution and the information provided by the emulation.

This approach allows the processor to execute the optimized code, while still providing *expected* behavior to the user; when execution is stopped in order to observe its state, the debugger will use partial emulation to make it appear as if all operations before the breakpoint have executed and all operations after the breakpoint have not. A major drawback to the approach taken is that it assumes conditional branches are never removed. Because synthesizing compilers regularly remove conditional branches through a process known as predication (see Section 3.2), these algorithms cannot be used by the hardware debugger.

The SC Debugger work takes an opposite approach to providing *expected* behavior, by making it appear that previous instructions have executed and future instructions have not. Rather than stopping early and emulating instructions, the hardware debugger buffers data and stops execution after all operations previous to the breakpoint have executed. *Expected* behavior is then retrieved by looking at the buffered data. Though the approaches are different, both require that the original ordering of operations is known. In both cases this is done by numbering the basic blocks in the original control-flow, then sorting instructions within a basic block by line numbers. Please see Chapter 9 for more information on how this is done in the hardware debugger.

2.2 Debugging of FPGA-based Applications at the Design Level of Abstraction

As FPGAs become larger, designers are starting to use higher level languages to describe circuits mapped to these devices. These languages can include hardware description languages such as VHDL [21] and Verilog [22], as well as languages based on

general purpose programming languages. For modern FPGA designs, hardware debugging is typically done at a different level of abstraction than the design, and even those that are debugged at the same level of abstraction are typically debugged in a different environment than the design. Some tools, such as Handel-C [23], provide software source-level debugging, but not hardware source-level debugging. Two projects that provide hardware debug in the design environment will be discussed in this section. The first allows limited debugging of circuits specified at the Register Transfer Level (RTL). The second allows debugging of circuit designed at the logical (structural) level. A third tool allows the designer to manually add debug hardware that can be controlled during hardware execution.

2.2.1 Identify [24]

The Identify tool from Synplicity, Inc. is an interesting new-comer to FPGA debugging tools. It is one of the first commercially available tools of which the author is aware that allows the user to specify signal watches at the design-level. In this case, the design-level of abstraction is the RTL level. The Identify tool works with Synplicity's synthesis tools to instrument designs to allow partial state of the executing circuit to be captured and displayed to the user. This is done by using on-chip resources to create triggers and buffers for capturing data while the circuit is run at full or near full speed.

The signals to be watched and used as triggers are specified by the user in the context of the original RTL description. This makes it very easy for the user to set watches. The information is provided to the user as waveform data, which is a typical way to debug an RTL level description. The drawbacks of the tool are that it provides only limited visibility, and that changing the values to be watched requires the circuit to be rebuilt, although the tool provides a mechanism to quickly rebuild the modified circuit.

2.2.2 Logical Hardware Debuggers for FPGA-Based Systems [25] by Paul Graham

This work is the first major research project that focused on enabling software-like debugging techniques for FPGA platforms. The main goal of this research was enabling complete visibility into the state of a circuit executing on an FPGA and presenting this data to the user in the context of the simulation environment used during design. It focused on designs created at the logical level. Much of the functionality developed in this research was added to the JHDL [26] CAD Suite.

Because this work was done for a low level of abstraction, it can be used as the basis for debugging circuits specified at a higher level of abstraction. Indeed, the current work on source-level debugging builds upon work done by Dr. Graham. This section will give a brief introduction to the methods used in providing the user complete internal state visibility of an executing FPGA circuit. This capability is enabled in two parts, discussed next.

The first part is a preprocessing step that creates a logical to physical mapping [27] for the application. This mapping defines a relationship between the values contained by elements in the logical description of the circuit and the physical location of that state in the hardware (see Figure 2.1). In JHDL, this mapping is provided in the `.rbentry` file, which associates JHDL circuit elements with the location of their values in the hardware state datastream. The state datastream can be provided by a number of different methods; Dr. Graham's dissertation specifically discusses the use of device readback [28] and design level scan [29], see Section 4.1.2 for more information on these two methods.

The second part of this project is a runtime interface that was built into JHDL that allows external values to be inserted into the JHDL simulation data structures. This interface is called the `ExternallyUpdateable` interface and allows state data (flip-flops, RAMs, ROMs, etc) extracted from the state datastream of the executing circuit using the logical to physical mapping to be inserted into the JHDL simulation data structures. This provides the user with a view of the values of the state elements in the circuit. To provide complete visibility of all signals in the circuit the simulator recreates the values generated

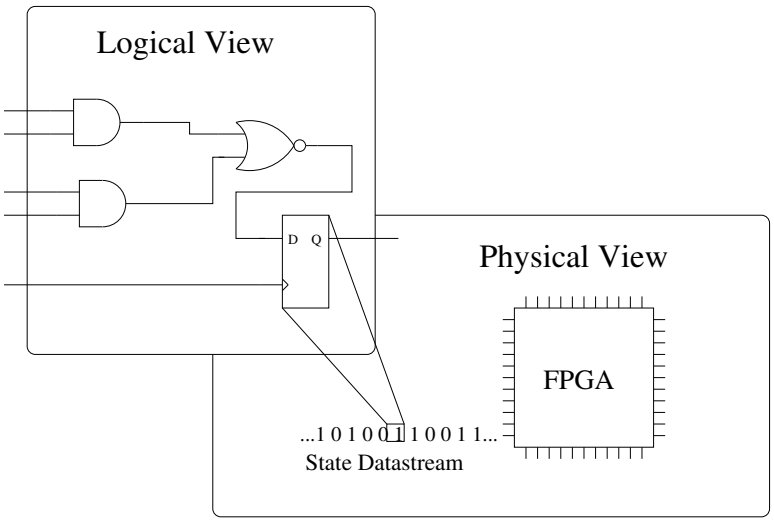


Figure 2.1: Logical to Physical Mapping

by combinational logic by simulating the appropriate circuit elements. This allows the user to view both synchronous and asynchronous signals in the design. This approach allows JHDL to provide the user with the same interface during both simulation and hardware debug.

Using JHDL as a Hardware Interface Layer

The Sea Cucumber Debugger, discussed in this work, uses JHDL as the interface layer to hardware. This has three main advantages:

1. Using JHDL as the interface layer allows the debugger to work in simulation as well as hardware mode. Thus, if the target FPGA platform is not currently available to execute the circuit, the debugger can still be used to view the simulating circuit in the context of the original source. The JHDL APIs used by the debugger are the same whether running in simulation or hardware.

2. Because JHDL has built-in tools to provide the logical to physical mapping, as well as runtime annotation of execution data, the work on the SC Debugger need only be concerned with mapping the source code to the logical level, rather than all the way to the physical level.

3. During hardware debugging, some signals of interest to the debugger may not be state elements, and therefore are not directly accessible in the hardware. JHDL is able to reproduce the state of these signals for the debugger's use.

2.2.3 ChipScope [30]

ChipScope is a tool provided by Xilinx for their Virtex and Virtex2 families of FPGAs. It provides debug components that can be included in a design written in either VHDL, Verilog or within the System Generator tool provided by Xilinx. The components are used to capture data from specific signals during hardware execution. The user decides at design time what signals are suspected of causing errors; these signals are then connected to the ChipScope components. At runtime, the ChipScope user interface allows the user to set trigger conditions for starting data capture. This data can then be displayed in a variety of formats. Though ChipScope provides a powerful debugging tool, it has the disadvantage of requiring manual instantiation by the user. Further, when using VHDL or Verilog, the debug information is displayed in the ChipScope environment, rather than in the context of the original source code.

When using the System Generator tool, the data can be manually exported from the ChipScope environment and imported into the System Generator environment. The main limitation is that only one ChipScope data capture component can be instantiated in a System Generator design. This severely limits the visibility provided by the tool.

2.3 Conclusions

There has been a large amount of research aimed at providing more efficient debugging capabilities for both optimized code and FPGA-based applications. The research described in this dissertation is working to combine these two areas to allow hardware debugging of synthesized code in the context of the original source code. As source level debugging of FPGA circuits exhibits many of the same difficulties as debugging optimized code, this work was greatly influenced by many ideas from this field of research.

Recent research in the area of FPGA hardware debug has provided knowledge of how to enable intuitive hardware debugging capabilities at the logical (circuit) level. This

research includes the work done by Dr. Graham, which provides insight into how to debug circuits specified at the structural level, as well as Identify, a new tool from Synplicity, which provides limited visibility for the hardware debugging of circuit specified at the RTL level.

Recent research has also lead to the creation of source level debuggers for high level languages, such as HandelC. However, these debuggers are only available for debugging the software description of the circuit, and do not allow for hardware debugging. This dissertation describes a source level debugger that moves beyond the debugging of the high level description, to the actual hardware debugging of the circuit. The debugger is used to report information about the state of the running circuit in the context of the original high level source. This is an important step forward, as debugging in hardware provides the advantage of debugging and verifying the final circuit rather than just the description of the circuit.

Chapter 3

Background

The research described in this dissertation concentrates on source level debugging of code which is synthesized using a VLIW-like compilation flow. Specifically, it focuses on debugging in the presence of predication, static single assignment (SSA), block merging and instruction scheduling. This chapter describes how these techniques are used to find exploitable parallelism in a program. Chapter 6 will discuss how these techniques, as well as other types of optimizations affect the hardware debug of the resulting FPGA circuits.

3.1 Static Single Assignment

Static Single Assignment [31, 32] is a transformation that produces an alternative representation of a program which has removed all false data dependencies in the entire program. The removal of false data dependencies allows the compiler more freedom during operation scheduling [33], leading to more exploitable parallelism. The remainder of this section will discuss the basic results of using SSA.

False dependencies are removed by allowing only a single assignment to a variable in a program. A program is transformed to SSA by creating a unique version of a variable each time a statement assigns it a value. Figure 3.1 shows a simple example of SSA. The example is simply a segment of code which increments the value of variable **a** several times. Note that each assignment to **a** is given a new version number. Uses of the variable after the assignment are changed to reference the new version.

For straight line code, as in Figure 3.1, converting code to SSA form is quite simple. The process is complicated by the presence of conditional branches and looping

```
a = 0;
a = a + 1;
a = a + 1;
a = a + 1;
```

(a) Original Code

```
a_1 = 0;
a_2 = a_1 + 1;
a_3 = a_2 + 1;
a_4 = a_3 + 1;
```

(b) SSA Code

Figure 3.1: Simple Example of Static Single Assignment

```
a = b + c;
if ( a > 10 ) {
  d = a;
}
else {
  d = 10;
}
e = d + b;
```

(a) Original Code

```
a_1 = b_1 + c_1;
if ( a_1 > 10 ) {
  d_2 = a_1;
}
else {
  d_3 = 10;
}
d_4 = mux(d_2, d_3);
e_3 = d_4 + b_1;
```

(b) SSA Code

Figure 3.2: Example of Static Single Assignment in Branching Constructs

constructs. In these cases, it is possible that a variable can be assigned along two different paths. When a variable is assigned along multiple paths, the compiler inserts a statement which can select the proper value based on the actual path taken. In the literature, these selection statements are referred to as ϕ-functions. In this work, I will refer to them as a mux-function, due to its resulting hardware implementation. The result of the mux-function is assigned to a new version of the variable as shown in Figure 3.2, which gives an example of converting to SSA form in the presence of branches. In this case the variable **d_4** is created to hold the result of selecting between **d_2** and **d_3**, which are assigned along opposite branches of the `if` statement.

In the case of looping constructs, the situation becomes more complex. In this case, one of the choices for the mux function will be a "future" value, i.e., one which is

```
i = b;
while ( <condition> ) {
  i = i + a;
}
c = i + b;
```

(a) Original Code

```
i_1 = b_1;
while ( <condition> ) {
  i_2 = mux(i_1, i_3);
  i_3 = i_2 + a_1;
}
i_4 = mux(i_1, i_3);
c_3 = i_4 + b_1;
```

(b) SSA Code

Figure 3.3: Example of Static Single Assignment in Looping Constructs

not yet assigned a value, but which will be in the body of the loop. An example of this is shown in Figure 3.3. Note that **i_2** chooses between **i_1**, which is assigned before the loop, and **i_3**, which is assigned during the loop. Further, it is interesting to note that an extra mux statement is needed after the loop because there is no guarantee that the loop body will ever execute.

When mapping SSA code to hardware, each of the variable versions is typically mapped to its own signal and each of the mux-functions is typically mapped to a 2:1 mux in the hardware. This is true of the Sea Cucumber (SC) synthesizing compiler (see Chapter 5). However, SC uses a slightly different, more restrictive formulation of static single assignment. In the SC formulation, the original code is broken up into basic blocks, and variable versions are made unique across these basic blocks. In addition, all variable values are passed between blocks through *primal* variables. The *primal* variables are unversioned variables, which are only used to initialize and store out values at the beginning and end of blocks. An example is given in Figure 3.4; the basic blocks are separated by blank lines.

This formulation limits the usefulness of SSA, as only false data dependencies within a single block are removed. However, when this formulation of SSA is combined with predication and hyperblock formation, discussed below, false data dependencies are removed across an entire hyperblock. Because SC only considers a single hyperblock during instruction scheduling, this formulation is sufficient.

```
a = b + c;
if ( a > 10 ) {

  d = a;

else {
  d = 10;
}

e = d + b;
```

(a) Original Code

```
b_1 = b;
c_1 = c;
a_1 = b_1 + c_1;
a = a_1;
if ( a_1 > 10 ) {

  a_2 = a;
  d_2 = a_2;
  d = d_2;
}

else {
  d_3 = 10;
  d = d_3;
}

d_4 = d;
b_2 = b;
e_3 = d_4 + b_2;
e = e_3;
```

(b) SSA Code using Primals

Figure 3.4: Example of Static Single Assignment Using Primal Variables

Because of the use of shared variables to pass values between basic blocks, there are no mux-functions in the original SSA code. During hyperblock formation, most of these primal references are changed to versioned variables and the mux-functions are inserted at this time (see Section 6.3.2). The results for the two formulations are much the same. The decision to formulate SSA using *primals* is based on the desired architecture of the synthesized circuit. Such a formulation allows a central repository for all variable values passed between blocks. SC calls this repository the thread registers [12].

3.2 Predication

Predication [34] is used to turn control-flow into data-flow. It does this by annotating instructions with the conditions under which they execute, these conditions are known as the predicates. This allows the target architecture to execute instructions along multiple branches, only committing the ones whose predicate has been met. This optimization allows the compiler to extract more instruction level parallelism from the application by allowing the compiler to group blocks of code together, reducing the amount of branching and increasing the potential number of instructions that can be executed in parallel.

The terminology in this work is slightly different than that typically used when speaking of predication and is adopted from the Sea Cucumber work. In this work a *predicate* is the boolean result of a comparison statement (i.e. the boolean result of $a > 10$). The execution condition for an instruction is known as the *predicate equation*, which is a boolean equation consisting of *predicates*. The *predicate equations* are determined using a process known as if-conversion [35, 34].

When assigning predicates, the original control-flow is broken into basic blocks. Transferring control from one block to another can be either conditional or unconditional. For conditional branches, there is a comparison statement that determines which branch is to be taken. As defined above, the result of these comparisons are *predicates*. Thus, each conditional branch in the control-flow can be annotated with a predicate (or inverted predicate), which gives information about the conditions for taking that branch. After all branches have been annotated, a predicate equation can be determined for each basic block. This is done by determining all possible paths to the block. As a path is traversed, each

conditional branch will contribute to the predicate equation. The final predicate equation is the union of the equations found for each path. Once the predicate equation for the block is found, all instructions in that block are annotated with that equation.

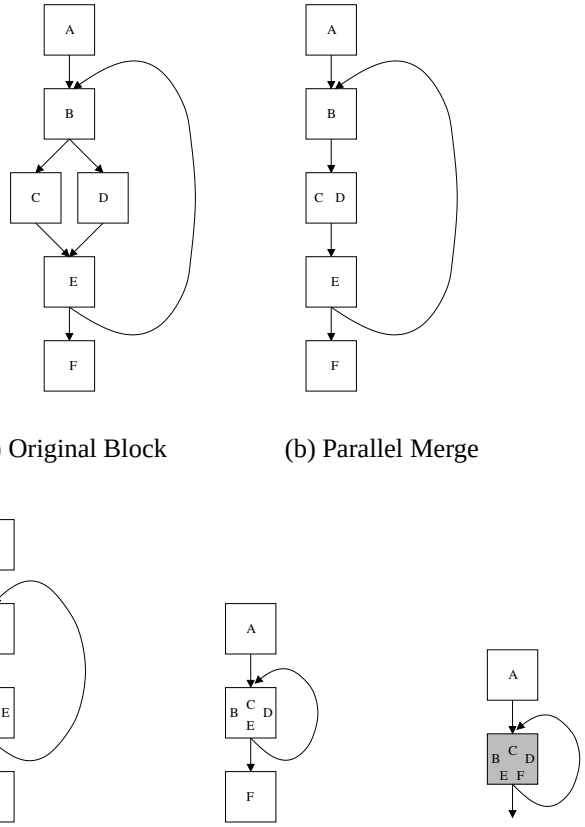
The result of predication is that every instruction in the program is annotated with a predicate equation, which gives the conditions that must be met for the result of the instruction to be committed. In implementations for predicated operations in general purpose CPUs [36], the predicate equations must be reduced to a single boolean value, thus instructions must be added to calculate the result of each equation. However, when using synthesizing compilers to create hardware, this limitation does not apply, as the equation can be computed using dedicated hardware.

3.3 Block Merging

After predicate equations have been assigned, the program is ready for block merging. Typically, the use of predication implies the use of block merging [35, 34, 37]. Block merging is the actual mechanism by which more instructions are made available for added parallelism. In this paper, it is assumed that when blocks are merged, only full blocks are merged together. This means that all instructions found in the same basic block in the original control-flow will be found in the same final block.

Though the techniques described in this work will generally work for any type of block merging (given the above constraint), this work concentrates on merging blocks to produce hyperblocks [37, 33]. Whereas the basic block is a set of instructions with one entry and one exit point, a hyperblock is a block with a single entry point, but any number of exit points. Though this restricts the size of a merged block, it is a good balance between parallelism and efficiency, and [37] claims that the use of hyperblocks improves the effectiveness of compiler optimizations.

When creating hyperblocks, blocks can be merged either serially or in parallel across all edges, except back edges. A block can be merged serially if it has a single in-edge in the control-flow graph, in which case it can be merged into the block from which this edge originates. Two blocks may be merged in parallel if they each have a single in-edge which originates from the same block and single out-edge which transfers control to



(a) Original Block (b) Parallel Merge

(c) Serial Merge (d) Serial Merge (e) Serial Merge

Figure 3.5: Example of Merging Blocks to Create a Hyperblock. The resulting hyperblock is shown in gray.

the same block. Figure 3.5 shows an example of merging several blocks together to create a hyperblock. Notice that the back edge into Block B makes it impossible to merge the previous block (Block A) into the hyperblock created by the other basic blocks. However, the back edge out of block E now becomes an early exit, which is allowed in the hyperblock.

3.4 Instruction Scheduling

Instruction scheduling is used to arrange the order of instruction execution to increase performance. There are many things which must be taken into account when scheduling instructions, and there are many different algorithms which can be used to do

this; a wide sampling of these algorithms is described in [38] and [39]. For the purposes of the current work, we will only consider two of the constraints on scheduling:

1. Data dependencies must be taken into account; a value cannot be used before it has been computed.

2. Instructions using the same resources cannot execute at the same time.

Under these two constraints, the scheduler attempts to schedule a set of operations to execute in the shortest time possible. This is typically done by executing as many operations as possible in parallel.

The result of instruction scheduling is that operations have been re-ordered and parallelized. The consequence of this is that the final flow of instructions is significantly different than the initial flow. In the current work, we assume that instruction scheduling only happens within a block (basic block, hyperblock or other merged block) and not across block boundaries.

3.5 Conclusions

Predication and static single assignment are common techniques used to enhance the available parallelism found in applications written in high level programming languages. Predication allows execution to proceed even when control-flow operations have not yet been computed, allowing the compiler to combine blocks of code together, thereby enabling more exploitation of parallelism. Static single assignment allows the compiler to schedule operations according to real data dependencies, not the dependencies imposed by shared memory locations. These optimizations make it possible to merge blocks and schedule the resulting larger group of instructions to take advantage of greater parallelism. Instruction scheduling results in reordered code which makes it difficult to intuitively display information about an executing program in the context of the original source code.

Chapter 4

Platform Requirements for FPGA Debug

In order to provide hardware debug capabilities, an FPGA platform must provide a certain degree of controllability and observability, including the ability to start and stop the circuit clock, and read and set circuit state. The way the platform provides this control is quite varied. In some cases, when the target platform does not provide the needed capability, the synthesizer can add hardware to the circuit to provide the necessary feature.

There is a recent body of research [25, 40, 29] which has outlined the level of controllability and observability which is needed to enable hardware debugging of modern FPGA-based platforms. This chapter will briefly review the main requirements necessary to enable hardware debugging of FPGA circuits and the ways in which the platform or compiler-added circuitry can provide the functionality. It will include a discussion of general functionality for all hardware debugging of FPGAs, as well as more specific requirements for enabling source level debugging of synthesized code. The chapter will close by discussing the Slaac-1V [41] configurable computing platform, which was used as a test bed for the SC Debugger.

4.1 Hardware Requirements

There are three basic requirements for enabling source level debugging of synthesized circuits. The debugger must be able to control the clock, and read and write the state of elements in the circuit. Each of these areas of functionality will be discussed.

4.1.1 Clock Control

Clock control is necessary to provide the debugger with both single stepping and breakpointing capability. The debugger must be able stop the circuit clock in order to read or set the state of variables. It is also advantageous, though not necessary, for breakpointing if the clock can be stopped directly from the circuit[1]. The debugger must also be able to both advance the clock a single clock cycle and allow the clock to free-run. The key to providing this feature is the ability to stop and restart the clock. In general, clock control should be provided as a feature of the FPGA platform.

If clock stopping is not supported on the target platform, then the synthesizer can add circuitry to the design to allow for this capability. For example, if using the Xilinx Virtex [42] series of FPGAs, then the built-in DLLs can be used to create a glitch-free clock stopping circuit [25]. A newer family of FPGAs, the Virtex II family [43], has an even more convenient method for providing clock control. The Virtex II FPGA provides the `bufgce` primitive. The `bufgce` is a "global clock buffer with a single gated input" [44]. When the clock enable on the `bufgce` is deasserted, then the output on the clock line is a logical "0". When the buffer is enabled, the clock is driven as normal.

4.1.2 Observing Circuit State

In order to provide any information about a running circuit, a hardware debugger must be able to observe the state of the running circuit. For source level debugging it is important to have complete visibility into the state of a circuit; limiting the visibility will limit the effectiveness of the debugger. This section talks about two methods of providing complete visibility of the state of an executing circuit and will discuss the advantages and disadvantages of each.

Readback

Many families of FPGAs provide the ability to capture the current state of the running hardware. This is provided as part of the readback [28] capability. Readback

[1] See Section 7.1 on how breakpointing can be accomplished with or without the ability of the circuit to stop the hardware clock.

allows the user complete access to the state of flip-flops and memories on the device. The major advantage of this approach is that the functionality is built into the device. However, not all platforms based on these types of devices give access to this feature. The largest disadvantage is the relative difficulty of determining which bits in the readback bitstream represent which state elements in the circuit. A solution to this problem for the Xilinx XC4000 and Virtex families of FPGAs was proposed in [25]. This solution has since been extended to work with Virtex 2 devices as well, and was described briefly in Section 2.2.2.

Scan Chain

If the ability to readback the state of an FPGA is not available, then it is possible to add the ability directly to the circuit. This is done by adding a design level scan chain [29] to the hardware. When adding design level scan, each state element of interest is added to a chain which can be scanned out of the device in order to get the state of the circuit. This is done by inserting a 2:1 mux in front of each flip-flop and connecting each flop to its neighbor as well as to its normal connection. The advantage of this approach is that only the bits of interest need to be added to the scan chain. The disadvantage of design level scan is that it can require a large amount of extra resources to implement. According to [45], adding design level scan can increase the size of the circuit by 60% to 80%, and at the same time, reduces the clock speed of the design.

4.1.3 Setting Circuit State

Another important feature required for debugging is the ability to set the state of circuit elements. In the source level debugger, this is used to set the value of variables. It can also be used to interact with the debug circuitry added to the circuit by the synthesizer (see Section 7.4.1). Unfortunately, there are no modern devices with a built-in way to externally modify the state of an FPGA circuit. However, there are two approaches that can accomplish this.

Scan Chain

The same scan chain that can be used to shift out values of state elements in a circuit can be used to scan new values back in. Of course, using scan chain in this way incurs the same area and speed penalties described above.

Bitstream Manipulation

Another way to provide the ability to set circuit state is through manipulation of the programming bitstream. In this case, the design is modified, such that the power-on state of each flip-flop is set to the desired state. After the modified bitstream is programmed into the device, all flip-flops will power-on into the specified state. The obvious advantage of this approach is that no additional hardware is needed to implement the state setting. However, there is a limitation on this approach when used with Virtex FPGAs, for example: The user cannot use the built-in set/reset features on flip-flops. This is because the power-on state of the flip-flop and the reset state are the same. Therefore, any flip-flop which has its state set has the potential of having its intended reset state changed. This limitation does not exist for Virtex II devices as the power-on and reset states are independent.

There are two main approaches to setting circuit state through the use of bitstream manipulation. This first approach, discussed in [25] uses a commercially available tool called JBits [46]. The advantage of this tool is that it is supplied and supported by the FPGA vendor. The disadvantage is the high cost and the fact that a non-disclosure agreement (NDA) must be signed in order to use the software. The second approach, used in [40], was to create a custom tool which had the ability to extract the current state from a readback bitstream and insert this state, or any arbitrary state, into the configuration bitstream. The advantage is that there is no monetary cost for the tool, and no NDA had to be signed[2]. The disadvantage is found in the time it takes to create the custom tool, and the fact that the library must be recreated for each new family of devices.

[2] Just before publication of this dissertation, the licensing for JBits was made more favorable, requiring no NDA

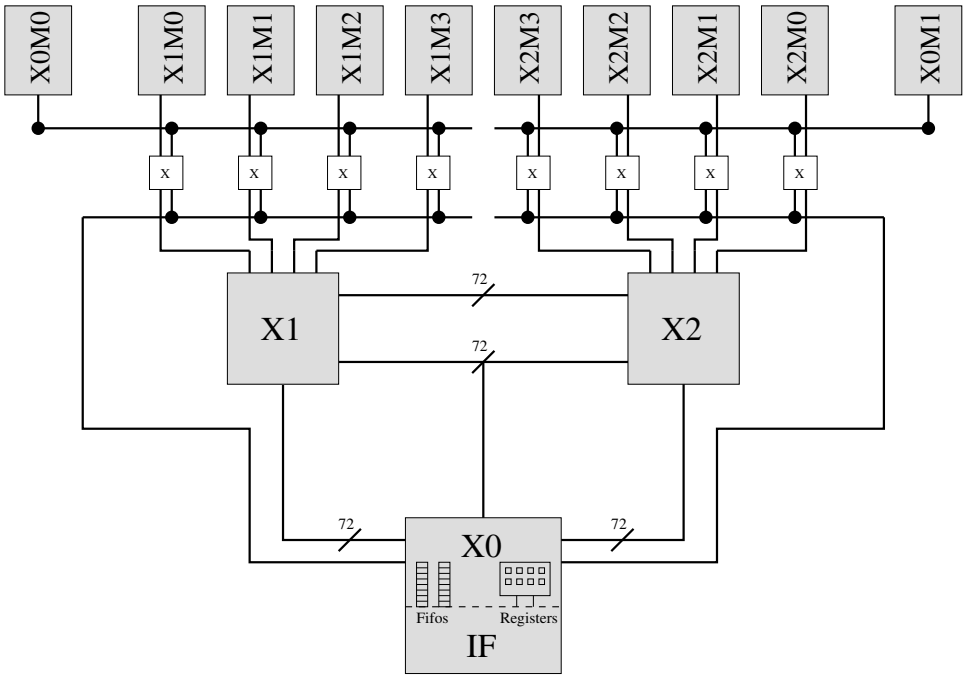


Figure 4.1: Block Diagram of the Slaac-1V Configurable Computing Platform

4.2 Slaac-1V

The Slaac-1V configurable computing board [41, 47] was used as the testbed for the SC Debugger. The Slaac-1V board was chosen because of the rich feature set that it provides. Figure 4.1 shows a block diagram of the Slaac-1V architecture.

As can be seen in the figure, the Slaac-1V is made up of three FPGAs, named X0, X1 and X2. The three FPGAs are connected through a ring which has 72-bit connections between X0 and X1, X1 and X2, and X2 and X0. These is also a 72-bit crossbar, which connects all three FPGAs. Both X1 and X2 are fully usable by the user; X0 contains both user circuitry and interface circuitry that is not user accessible. Included with the interface logic in X0 are FIFOs and a register interface that allows software read/write access to eight 32-bit registers. These registers are used by the debugger to control the global debug signals which will be discussed in Chapter 7. The SC Debugger places the synthesized design in X1. Because of this, the register interface signals must be passed through X0 to X1.

The Slaac-1V also provides other important features necessary to enable run-time debugging, such as the ability to configure and readback the state of each of the FPGAs. Also, the Slaac-1V board was the test platform for the bitstream manipulation tool described in [40]. This gives the ability to both read and write the state of the circuit.

In addition, the Slaac-1V provides a high degree of clock control to the user. The clock can be single-stepped, stepped a specified number of cycles, free-run and stopped. Additionally, the user circuitry can request that the clock be stopped. However, it takes multiple cycles to stop the circuit clock in this manner.

The Slaac-1V board also provides many other useful features, such as host read/write access to board level memories, but these are not pertinent to the work at hand, so will not be discussed here.

4.3 Conclusions

Enabling hardware debugging capabilities requires the target platform to provide basic controllability and observability functions. If the platform does not provide a particular feature, it falls to the compiler to add the necessary hardware to enable debugging. While the addition of this hardware will effect circuit area and speed, the debug circuitry can be removed after debugging to enable the highest possible circuit performance. It is desirable to have the target platform directly support all necessary debugging features, but the reconfigurability of FPGAs allows features to be added and deleted as needed.

Chapter 5

Sea Cucumber

To provide a test bed for the current debugger work, I created a debugger for the Sea Cucumber (SC) synthesizing compiler [10]. Sea Cucumber was chosen because it is representative of compilers which use predication and static single assignment to find exploitable parallelism, and because I had access to read and modify the source code. I worked closely with the creators of Sea Cucumber and modified SC to provide information about the synthesis process. This information is contained in the Debug Database (described in the next chapter). The SC Debugger provides an implementation of most of the ideas presented in this dissertation. In order to provide a better understanding of the examples in later chapters, this chapter will describe the Sea Cucumber compiler.

Sea Cucumber provides a programming model and a circuit synthesis tool. The programming model provides a framework within which programmers can express their designs. This design is used as input to the circuit synthesis tool, which generates a JHDL circuit that can be both simulated in the JHDL simulator and netlisted to EDIF [48] for execution on an FPGA. The goals of the Sea Cucumber synthesizing compiler include the exploitation of a high degree of parallelism and software executable behavior that matches hardware execution as nearly as possible.

This chapter will separately discuss the programming model and the synthesis engine. Both of these pieces are important to the understanding of the operation of the Sea Cucumber Debugger.

5.1 Programming Model

The SC programming model specifies the type of Java code which can be synthesized by Sea Cucumber. This model is made up of two pieces: Java threads and Concurrent Sequential Process (CSP) channels [49]. This programming model allows programmers to express coarse grain parallelism through the use of Java threads. Fine grained parallelism within a thread is extracted by the Sea Cucumber compiler, and inter-thread communication is accomplished though the use of CSP channels. The software behavior of these channels is provided in a library and allows the programmer to simulate and debug the description of the design using the Java Virtual Machine (JVM) and normal software debuggers. The following sections will describe the use of threads and channels in greater detail.

5.1.1 Threads

The thread programming model is a well known model [50] for specifying program level parallelism. Threads allow a program running on a general purpose computer to give the illusion of concurrency. On a single processor system, the operating system will schedule threads to run one at a time on the hardware, though to the user the threads will appear to run concurrently. The use of multiple processor systems allows a limited number of these threads to run simultaneously. However, when SC maps the design to a circuit, all threads will become dedicated hardware that is guaranteed to execute in parallel (subject to the constraints placed on the application by the use of synchronization points).

To program a thread in the SC framework, the programmer simply creates a class which extends `Thread`. The programmer then overloads the `run()` method to add the desired behavior of the thread. This is the common Java paradigm for using threads [51]. Figure 5.1 illustrates how a typical SC thread class might appear.

Though Sea Cucumber uses typical Java techniques for determining the behavior of threads, it does limit the ways in which threads can communicate with one another. A thread is only allowed to read values from or write values to its own variables. The Sea Cucumber compiler will not allow one thread to read or modify another thread's variables.

```
public class Adder extends Thread {
  public Adder() {
    // Call start() to begin thread execution
    start();
  }

  // Overload run method to add thread's behavior
  public void run() {
    int a = 2;
    int b = 3;
    int c;

    c = a + b;
  }
}
```

Figure 5.1: Example of How to Create a Thread in Sea Cucumber

The only way to pass data between threads is through the use of channels. Channels are discussed in detail in the next section.

5.2 CSP Channels

The inter-thread communication model for Sea Cucumber is based on CSP channels. These channels provide and manage communications between threads. Basic CSP communications are unbuffered[1], one-way and self-synchronizing. In order for communication to occur, both sending and receiving threads must be ready. When one thread reaches the communication point first, it will block until the other thread arrives to complete the communication. This allows channel communication to act as a synchronization method.

It has been proven [49] that the use of CSP channels provides predictable and consistent thread behavior. This does not mean that the use of CSP channels will not lead to thread deadlock. However, it does mean that the order and combination of thread execution

[1]While CSP channels are technically unbuffered, it is possible to build up buffered communication schemes using the simple unbuffered communication primitives. See [52] for more information about the types of channels supported in Sea Cucumber.

will not effect the onset of deadlock. In practice this means that for the same data set, a program will either deadlock every time it is executed, or not at all. This predictable behavior allows the programmer to concentrate on the behavior of the threads, rather than on the communications.

Sea Cucumber provides a library that implements the behavior of CSP channels. The software behavior of these channels matches the behavior of the hardware generated for inter-thread communications. A detailed discussion of the syntax for using channels in the Sea Cucumber framework can be found in [52]. Figure 5.2 shows the basic additions to the thread code in Figure 5.1 needed to use channels.

5.2.1 Relationship Between Threads and Channels

The relationship between channels and threads is specified in the `main()` method of the primary class. The `main()` method describes what type of threads and channels are used and how they are connected to each other. An example is shown in Figure 5.3.

The example `main()` method creates three threads (input, adder and output) and connects them as shown in Figure 5.4. The threads and channels are created using object creation. The channels are typically created first and are then passed into the threads as arguments to their constructors.

5.3 Circuit Synthesis

The Sea Cucumber synthesizer uses a VLIW-like[2] compilation flow to generate optimized hardware for the input application. The inputs to the synthesizer are the Java class files obtained by compiling the Java source files used to describe the application. These class files are used both for the original execution of the application in software, as well as the starting point for synthesis. The output of this process is a JHDL [26] circuit built in memory and used to netlist EDIF or other targets supported by JHDL. Figure 5.5 shows the basic steps used by Sea Cucumber during the synthesis process: bytecode

[2]There are different approaches to VLIW compilation; SC uses a flow similar to that used for VLIW machines which use predication.

```
public class Adder extends Thread {
  // References to input and output channels
  private Input a_in;
  private Input b_in;
  private Output out;

  // Channels must be passed in through the constructor
  public Adder(Input a_in, Input b_in, Output out) {
    // Keep references to the channels
    this.a_in = a_in;
    this.b_in = b_in;
    this.out = out;
    // Call start() to begin thread execution
    start();
  }

  // Overload the run method to add thread's behavior
  public void run() {
    int a, b, c;

    while(true) {
      // Read values from input channels
      int a = a_in.getInt();
      int b = b_in.getInt();

      c = a + b;
      // Write result to output channel
      out.putInt(c);
    }
  }
}
```

Figure 5.2: Example of Using Channels in Sea Cucumber Threads. This example includes the extra channel code for the example in Figure 5.1. The thread continually reads values from the two input channels, adds them together, and writes the result to the output channel.

```
public static void main(String[] args) {
  // Create the channels
  IntBlockingChannel a_in = new IntBlockingChannel();
  IntBlockingChannel b_in = new IntBlockingChannel();
  IntBlockingChannel out = new IntBlockingChannel();

  // Now create threads and pass in channels appropriately
  InputThread input = new InputThread(a_in,b_in)
  Adder adder = new Adder( a_in, b_in, out );
  OutputThread output = new OutputThread(out)
}
```

Figure 5.3: Example of a Simple Sea Cucumber `main()` Method. The method is used to create channels and threads and describes how they are connected to each other.

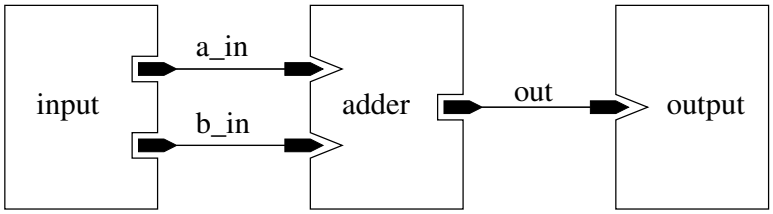


Figure 5.4: Relationship Between Threads and Channels. The threads and channels are described in the `main()` method found in Figure 5.3

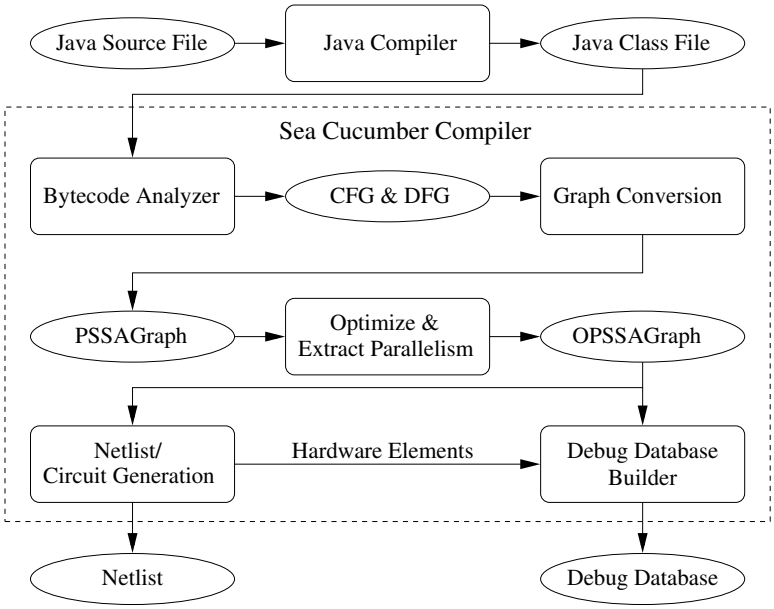


Figure 5.5: Overview of Operation Performed by Sea Cucumber. This shows the basic steps performed by Sea Cucumber during the synthesis process, including operations added to aid in debugging. Rectangular nodes represent operations or tools; oval nodes represent data files or formats.

analysis, graph conversion, extraction of fine-grain parallelism, compiler optimizations and circuit generation. Each of these is discussed in detail in the following sections. An example of the major steps in the compilation process are shown in Figure 5.6. This example is used in the following sections.

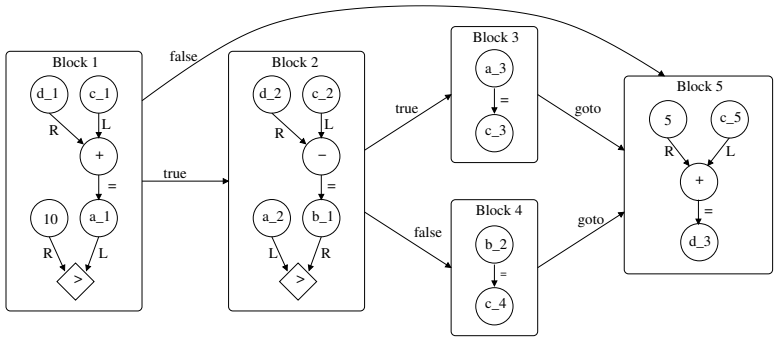
5.3.1 Bytecode Analysis

The first step in the Sea Cucumber compilation flow is bytecode analysis. The low-level bytecode parsing and initial analysis is done by Soot [53], a Java optimization framework. Soot provides an intermediate representation of the Java bytecode. The information is used to produce control-flow (CFG) and data-flow graphs (DFG) of the application. Each node in the CFG contains a DFG that represents a basic block.[3] During the creation of the DFGs, static single assignment is introduced by creating a new version of

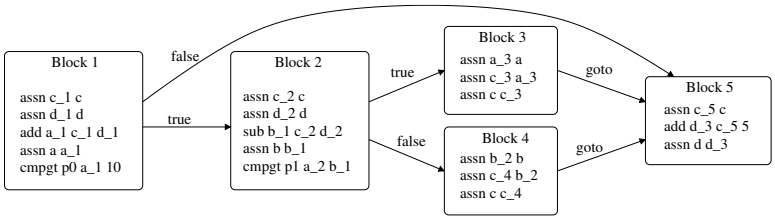
[3]A basic block is a block of sequential instructions with a single entry and a single exit point [32] Basic blocks are bounded by branching statements such as if-then-else structures and loops.

```
39 a = c + d
40
41 if ( a > 10 ) {
42   b = c - d;
43
44   if ( a > b )
45     c = a;
46   else
47     c = b;
48 }
49 d = c + 5;
```

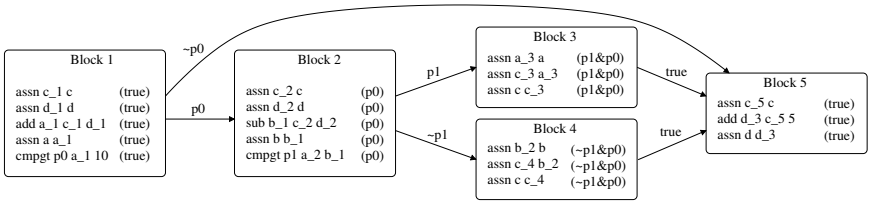
(a) Source Code



(b) CFG with DFG Nodes



(c) CFG with Operation Lists



(d) CFG with Predicated Operation Lists

```
0  add a_1 c d                          (true)
0  sub b_1 c d                          (p0)
1  cmpgt p0 a_1 10                      (true)
1  assn a a_1                           (true)
1  cmpgt p1 a_1 b_1                     (p0)
2  mux c_6 a_1 b_2 <~p1&p0>             (p0)
2  assn b b_1                           (p0)
3  mux c_7 c_6 c <p0>                   (true)
3  assn c c_6                           (p0)
4  add d_3 c_7 5                        (true)
5  assn d d_3                           (true)
```

(e) Final Scheduled and Optimized Hyperblock

Figure 5.6: Overview of SC Compilation Flow. The example shows the major steps in the Sea Cucumber compilation flow.

a variable each time a new value is assigned to it. As described in Section 3.1, SC uses primal variables to pass variable values between basic blocks. A code segment and its accompanying CFG and DFGs are shown in Figures 5.6(a) and 5.6(b).

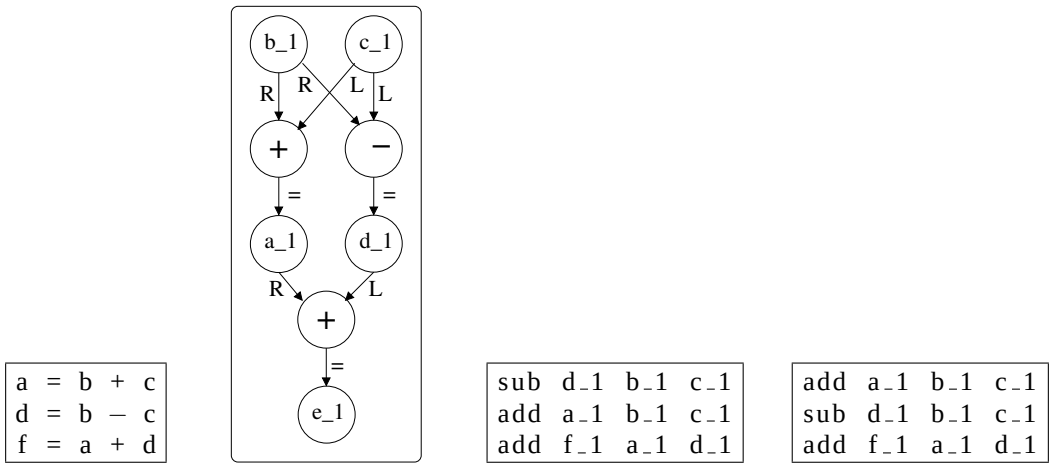
Soot is also used to extract other information about the design. It is used to parse the `main()` method used to describe the application at the top level as discussed in Section 5.2.1. This information is passed to SC to determine how threads and channels are connected. Soot is also used to extract debug information from the class file symbol tables[4] that contain information about variable names and line numbers from which operation are derived. The variable names are used to correctly label the variables in the SC graph structures. Each operation in the original DFG is annotated with the line number from which the operation was derived. This information is required by the SC Debugger.

5.3.2 Graph Conversion

In graph conversion, the DFGs and CFGs created during bytecode analysis are converted to the SC internal graph format, the `PSSAGraph`. The `PSSAGraph` is a CFG where each node of the graph contains a list of operations. The original list of operations is derived from the DFG for each node in the original CFG. This can be seen in the example in Figure 5.6(c).

Ideally, the operation lists created during this step would identically mirror the instruction ordering in the original source code. However, this is not the case. The DFGs created in the previous step preserve only operation dependencies, not operation ordering. In the example shown in Figure 5.7(a), the assignment to variable a clearly happens before the assignment to d. However, the DFG (Figure 5.7(b)), shows that there is no data dependence of d on a. Therefore, when the graph is traversed to be converted into an operation list, either of the lists found in Figure 5.7(c) is a possible outcome of the conversion process. This means that the operation lists created in this stage do not necessarily reflect the original instruction ordering in the source code. The consequences of this loss of sequencing information is discussed in Section 6.3.

[4] Section 6.1 contains a detailed discussion of Java symbol tables.



(a) Source Code (b) DFG (c) Two Possible Operation Lists

Figure 5.7: Example of DFG Losing Information about Instruction Ordering

5.3.3 Fine-grained Parallelism Extraction

Fine-grained parallelism is extracted from the `PSSAGraph` by manipulating the graph and converting maximal control-flow into data-flow. This is done through the creation of hyperblocks. Operation predication allows an operation to be annotated with information about whether its result is valid and allows operations to be executed and later invalidated, if necessary. Figure 5.6(d) shows the operation lists in Figure 5.6(c) annotated with predicate information. If-conversion is used to compute this predicate information. Using these techniques, SC transforms control-flow into data-flow, increasing the number of operations that can be in a single block. These larger blocks create a greater potential for executing operations concurrently, enhancing the available fine-grained parallelism.

Once all operations are annotated with predicate information, basic blocks are merged into hyperblocks. Edges in the `PSSAGraph` are separated into two different groups: forward edges and backward edges [54]. Blocks may be merged in parallel or in serial across all edges except backward edges, as discussed in Section 3.3. In practice, this means that hyperblocks are bounded by loops. Serial and parallel merges are described

in more detail in Section 6.3.2. After merging is complete, each node in the `PSSAGraph` represents a hyperblock.

5.3.4 Compiler Optimizations

Before synthesis, SC optimizes the user's design to reduce the hardware requirements of the design. These optimizations happen at different levels in the `PSSAGraph`: graph level, hyperblock level, and operation level. The optimizations performed by SC, as well as how they effect debugging, are discussed in some detail in Section 6.3.2.

The final optimization performed is instruction scheduling. SC schedules operation by dependence in an "as soon as possible" (ASAP) manner [39]; operations are scheduled as soon as their inputs are ready. The scheduler assigns each operation to a given execution cycle. Figure 5.6(e) shows the final optimized and scheduled hyperblock for the code segment in Figure 5.6(a). The cycle information is used to form a logical state machine for each hyperblock in the graph. These state-machines control the execution flow within a hyperblock, and are also used to control when and which outputs are written to registers.

5.3.5 Netlist Generation and Synthesis

During netlist generation, each thread, represented by a `PSSAGraph`, is converted into a generic circuit object. Because the operations in a hyperblock are quite simple (i.e. add, subtract, shift, multiply, etc.), they lend themselves to direct synthesis into the circuit. Other elements are handled as special cases: thread registers, state-machines and hyperblock control, memory elements, and CSP channels.

The primal values passed between hyperblocks are kept in the thread registers. Each thread has its own set of thread registers. When the hyperblock is synthesized, any required variable not generated within the hyperblock is read from the thread registers. Any value needed by future hyperblocks is written to the thread registers after it is computed.

State machines are generated from their logical descriptions created after instruction scheduling. The state signals are used to control writing of registers within the

hyperblock and to the thread registers. The state machines also use the predicate information to determine the control-flow between hyperblocks; the currently active hyperblock will start the state machine for the next hyperblock (which may be itself) when it completes execution.

The CSP channels are not synthesized, but are rather hand mapped hardware, which is inserted into the circuit as necessary. For more information on what this hardware looks like see [52]. The channel hardware was written such that its synchronization behavior exactly matches the software behavior. This is accomplished by using state machines on both the read and write sides of the channel to ensure that both threads block until the transfer is complete.

As stated previously the results of synthesis and netlisting is a JHDL circuit. This circuit can be used for a gate level simulation of the application, as well as for creating an EDIF netlist. The JHDL circuit can also be used to control the circuit executing in the FPGA. This feature is used by the debugger as described in Section 2.2.2.

5.4 Summary and Conclusions

Sea Cucumber uses well known VLIW compiler techniques to expose maximal parallelism, which can be exploited to its fullest in the final FPGA hardware. This compilation flow significantly changes the control- and data-flow of the original program, which makes it an interesting test case for the debugging work presented in this dissertation.

Chapter 6

Debug Database

In order for any debugger to operate, the compiler must provide information about how the source code is mapped to instructions. For software debuggers, this information is generally called the symbol table and consists of two basic parts: the line number table and the variable table. In order to avoid confusion, the debug information generated by the synthesizing compiler is called the debug database. It contains mappings similar to those found in a software symbol table, but in general requires much more information. As a comparison, the basic contents of a software symbol table, as they pertain to this work, will be discussed in the next section. This chapter will also look at the mappings contained in the debug database and how predication, static single assignment, block merging and instruction scheduling affect them. The chapter will end with specific examples of how information for the debug database is created and stored in Sea Cucumber.

6.1 Software Symbol Tables

Most software symbol tables contain similar information. This information is used to identify the line numbers from which the final object code originated, as well as find the location of variables specified in the source code. As an example, this section will discuss the contents of the symbol table found in Java class files. This information is also important to this work, as the information in the Java class files is the starting point for the creation of the hardware debug database for Sea Cucumber. When full debug support is turned on in the Java compiler (using the `-g` option), the compiler adds three pieces of information to the class file. The first piece of information is simply the name of the Java source file from which the class file was derived. The other two sets of information are the

line number table and the variable table. The contents and importance of these tables, as well as how Sea Cucumber uses this information, will be discussed in the next sections.

6.1.1 Line Number Table

The purpose of the line number table is to allow the debugger to determine from which line of source code each instruction in the executable code was derived. This allows the debugger to provide both single stepping and breakpointing capabilities, by allowing the debugger to determine the offset (or program counter value [55]) within the program of the first instruction for each line of source code. This offset is used to stop execution at the proper instruction for both single stepping and breakpointing.

In Java, each method compiled as part of a class file contains its own line number table. The line number table consists of a listing of those line numbers which contain valid instructions, along with the bytecode offset (from the beginning of the method) corresponding to the first instruction derived from that line. Because the Java compiler does not generally optimize the code during compilation[1] every line is clearly separated from every other line in the resulting bytecode. As an example, consider line 7 from the code in Figure 6.1. The line number table reports that line 7 begins at byte offset 5 and that line 8 begins at byte offset 9. This means that all operations starting at offset 5 inclusive and ending at offset 9, exclusive ($5 \leq$ `byte offset` < 9), are derived from line 8 of the source code.

Sea Cucumber, through Soot, uses the information in the line number table to annotate the operations in the original data-flow graphs with the line numbers from which the instructions were generated.

6.1.2 Variable Table

The purpose of the variable table is to allow the debugger to find the storage locations for the variables in the program. This information is used to view and change the value of the variables.

[1] If code is optimized by the Java compiler, then SC is able to start synthesis from the optimized program. However, these optimizations must be turned off to enable debugging, as the Java symbol table has no mechanism for providing mappings for optimized code,

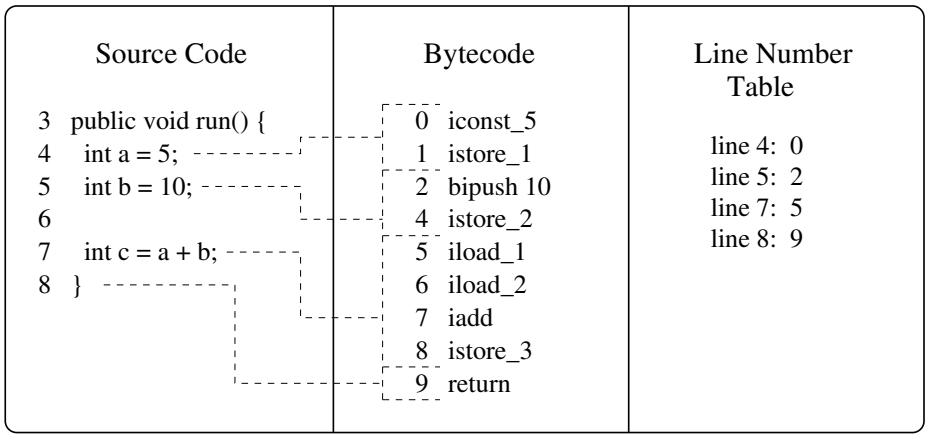


Figure 6.1: Example of a Java Line Number Table. The left column in the source code represents line numbers. The left column in the bytecode represents offsets in the bytecode. The table was found by executing `javap -c -l` on the class file.

Just as with the line number table, each Java method in a class file contains its own variable table. Each method also has a local variable array [56]; each variable defined in the method is mapped to a location in this array. The variable table maps slots in the local variable array to names and scopes within the method. The scope of a variable is specified by the byte offset at which the variable is first defined and the length of bytes in the code to where the variable ceases to be valid. An example of a Java Variable Table is found in Figure 6.2.

Java debuggers use the variable table to determine if variables are currently in scope, and if they are, it gives their storage location so that the values can be viewed and/or changed.

Sea Cucumber uses only the variable names in the variable table, not the scoping information, as it is not valid after SSA is introduced to the code. This information is used to properly name the variable nodes in the original data-flow graphs. This means that the variable names used by SC match the names in the source code wherever possible. Stack variables, which are not specified explicitly in the source code, are given synthetic names by the bytecode parser.

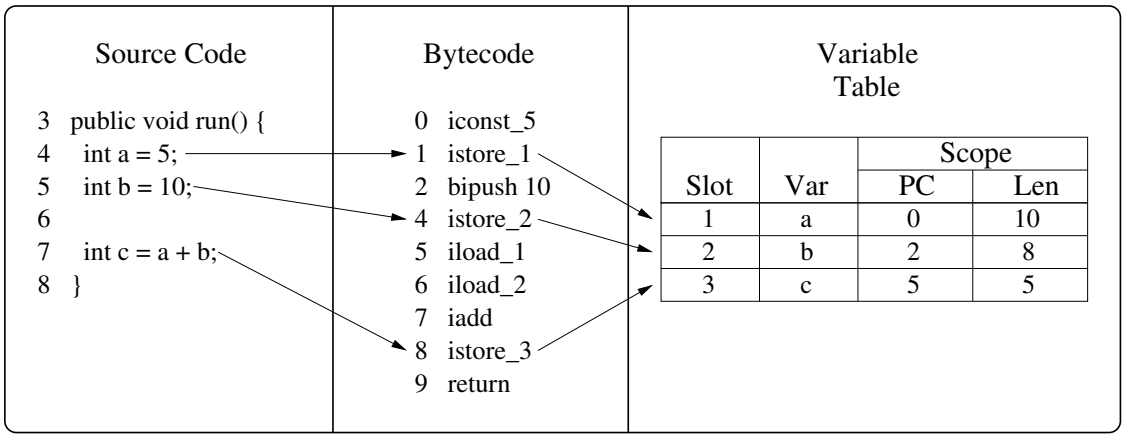


Figure 6.2: Example of a Java Variable Table. The left column in the bytecode represents offsets in the bytecode. The variable table shows the slot number in the local variable array, variable name, byte offset where existence starts (PC) and number of bytes of existence (Len). The arrows in the figure show the path from variable assignment in the source code to the corresponding assignment in the bytecode, to the entry in the variable table. The table was found by executing `javap -c -l` on the class file.

6.2 Hardware Debug Database

The hardware debug database contains the same type of information as the software symbol table, but is generally more detailed. The hardware debug database must enable the debugger to account for control-flow (roughly equivalent to the software line number table) and data-flow (roughly equivalent to the software variable table) in the final circuit and relate it back to the original source code. Though the debug database provides mappings similar to the symbol table, the mappings provided by the debug database are much more complex. This added complexity arises for two primary reasons: Lack of a fixed architecture and the presence of optimizations which dramatically change the control- and data-flow of the application. These issues are discussed in the following paragraphs.

Unlike a compiler for code mapped to a general purpose processor whose output is a list of instructions, a synthesizing compiler's output consists of a specialized circuit for each application. Because the output of the synthesizing compiler is not simply a list of instructions, the contents of the debug database cannot be based on offsets in the object code, locations in memory, etc. The final mappings must relate the source code to the

execution state of the resulting circuit. Thus, the mappings in the debug database lack a simple reference point like the start of the object code.

Another issue adding to the complexity of the mappings in the debug database is the presence of optimizations which transform the control- and data-flow of the program. These types of optimizations make it difficult to correlate the original source code with the reordered and optimized final program. The main difficulties arise because of the use of predication, static single assignment, hyperblock formation and instruction scheduling. However, other optimizations can pose problems in creating this mapping.

The two modes of debugger operation require different types of mappings; this means that there are potentially a large number of mappings from source code to hardware that need to be stored in the debug database. To simplify these mappings, the debug database uses incremental mappings between the three levels of abstraction shown in Figure 6.3: source level, CFG/DFG level and hardware level. The source level of abstraction is the level at which the debugger communicates information to and receives control instructions from the user. It consists of line numbers and variable names. The CFG/DFG level of abstraction is representative of the data structures used by the compiler and contains information about operations, SSA variables (operands), and schedules. The hardware level consists of circuit states and circuit elements such as registers, memories, wires, etc. This is the level at which the system observes and controls the running circuit. In addition to the mappings between these levels of abstraction, there are also incremental mappings found within the CFG/DFG level of abstraction. The incremental mappings can be reused by chaining them together in different ways to produce the final mappings required by the debugger.

The next sections will discuss the general information and mappings needed in the hardware line number and variable tables. They will also discuss the issues arising from the use of predication and instruction scheduling (which affect the line number table), static single assignment (which affects the variable table) and hyperblock formation (which can affect both tables), and how these optimizations drive the need for additional information in the debug database. Also discussed will be some general solutions to extracting and

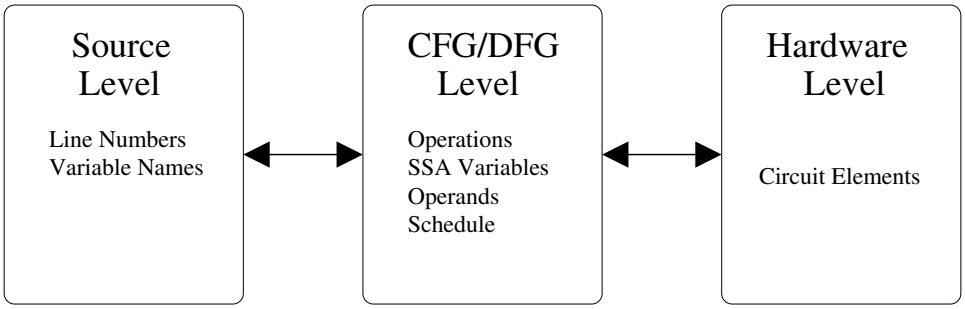


Figure 6.3: Levels of Abstraction Used in the Debug Database.

providing this information. The specific solutions used in the Sea Cucumber Debugger will be discussed in Section 6.3.

6.2.1 Accounting for Control-Flow: The Hardware Line Number Table

Similar to the software line number table, the hardware line number table provides the debugger with the information necessary to observe and control the execution of the application. In order to do this, the hardware line number table is required to relate the original control-flow described in the source code to the final control-flow of the optimized circuit. As explained above, this mapping is complicated by predication, hyperblock formation and instruction scheduling.

Predication and Hyperblock Formation The process of creating hyperblocks transforms some of the control-flow found in the original program into data-flow; the debug database must account for this information. When instructions are predicated and blocks are merged together, it becomes possible to speculatively execute operations. This means that operations may execute and later be invalidated, or execute and not have their results committed for future use. This makes it more difficult to determine, at any given instant, which of the executing operations is contributing to the results of the application. In order to sort this out, the debug database must store information about the predicate equations for each operation in the optimized application. This information allows the debugger to determine which results will be committed for future use. However, because there is no

requirement that an operation's predicate equation be computed before executing that operation, it is entirely possible that there is no way to tell if an operation will be committed at the time of its execution. This has major implications in how the debugger operates. This will be discussed in more detail in Chapters 8 and 9.

Instruction Scheduling One of the reasons for the added complexity of the hardware line number table lies in the fact that the scheduling of computations is not explicitly given by the circuit description. Unlike a software executable, where the operations are assumed to execute in the order in which they appear in the executable,[2] it is not easy to determine operation ordering by looking at the resulting circuit. Because the hardware representation is not simply a list of instructions to process, the scheduling information for the instructions is not explicit. Therefore, this information must be computed and inserted into the debug database. In hardware the "program counter" is typically comprised of the aggregate state of all control units in the hardware. So, instead of mapping simple offsets to line numbers, the hardware line number table is required to store the information that shows the relationship between the state of the control circuitry and the line numbers of a source file. The way this relationship is established can be quite different depending on the synthesizer.

Incremental Mappings in the Hardware Line Number Table

In order to provide the mappings required to allow the ability to observe and control the execution of the application, the hardware line number table makes a series of incremental mappings which can be used to create the final full mappings. It is important to note that some of these mappings are two-way mappings (denoted by $\longleftrightarrow$) and others are only one-way mappings (denoted by $\longrightarrow$). A list, along with explanations, of these mappings is given below. Since the use of the full names of the mappings is unwieldy, each mapping has been given a number (L1-L7). These numbers will be used when referencing the incremental mappings. However, since remembering the numbers can be difficult, each

[2] This does not strictly hold true for superscalar machines. However, the superscalar machine typically uses a reorder buffer to allow operations to be committed in the proper order. Thus, even though internal to the machine the instructions are executing out of order, the ordering of the results from the perspective of an observer is unchanged.

mapping will also be given a mnemonic to make it easier to determine which mapping is being referenced. The mnemonic for each is given after the name of the mapping. For example the first mapping will be referred to as L1:Line-Op.

L1. **Source Line $\longleftrightarrow$ Original Operation (Line-Op).** This two way mapping is used to correlate line numbers in the source code to operations in the original DFG or operation list. Given a source line, this mapping gives a list of all original operations which were derived from this line of code. The reverse mapping maps an original operation to the line of code from which it was derived. The reverse mapping may be empty as some operations introduced by the compiler do not correlate directly to a line of source code. For example, reads and writes to thread registers introduced during compilation by SC do not correspond to any instruction in the original source code.

L2. **Original Operation $\longleftrightarrow$ Initial Schedule (Op-InitSched)** This two way mapping provides the debugger with the ordering of the operations in the original control-flow. In this case, the schedule consists of the basic block that contains the operation, as well as a number which gives the order of the instructions within the block. This mapping is only needed when operating the debugger in source step mode, and is used as a baseline for sequentializing the operations in the final parallelized circuit. Since this schedule gives the ordering of the original schedule, this mapping must also contain information about the control-flow between basic blocks.

L3. **Original Operation $\longleftrightarrow$ Final Operation (OrigOp-FinalOp).** This two way mapping is used to map the original operations extracted from the program to the final operations in the fully optimized and scheduled application. The forward mapping provides the final operations which have encapsulated the functionality of the original operation. If this mapping is void then this means that the original operation was removed during optimizations. The reverse mapping gives the original operations which are encapsulated in the specified final operation. A void reverse mapping means that the operation was inserted by the compiler to ensure the correctness of the program.

L4. **Final Operation $\longleftrightarrow$ Schedule (Op-FinalSched).** This two way mapping is used to correlate operations in the final graph with their final logical schedules. The logical

schedule is made up of the hyperblock in which the operation is found, along with a number representing the state during which the operation executes. The forward mapping maps an operation to a specific time in the schedule. The reverse mapping is used to find all operations which execute at the specified time.

L5. **Operation $\longrightarrow$ Predicate Equation (Op-Pred).** Because predication allows operations to be executed but later invalidated, it is necessary for the debug database to be able to map an operation to its predicate equation. This provides information on when the result of an operation will be committed for later use. This mapping simply provides the predicate equation for each operation found in either the original or final control-flow. While it is possible to create a reverse mapping that would provide a list of instructions which have the specified predicate equation, this information is generally not necessary for the debugger to function.

L6. **Schedule $\longleftrightarrow$ Circuit State (Sched-State).** This mapping is used to correlate the logical schedule with the equivalent state of the control circuitry. The forward mapping provides information about the circuit state which corresponds to the specified logical schedule. The reverse mapping takes the state of the circuit and determines the current position in the logical schedule. The exact way in which this mapping is established is dependent on how the synthesizer generates the control for the circuit.

L7. **Operation $\longrightarrow$ Breakpoint Unit Programming Data (Op-BPU).** The debug database is required to store information about the circuitry added to enable hardware breakpoints. This circuitry is referred to as a breakpoint unit[3], and can be programmed at runtime to stop circuit execution at specified times. This one way mapping gives the debugger the data stream needed to program the inserted breakpoint units to stop circuit execution when the specified instruction executes.

[3]For more information on the breakpoint unit inserted into the synthesized circuit see Chapter 7.

General Data Structures Used to Provide Incremental Mappings

This section will discuss a possible general form for data structures which can be used to efficiently store the above mappings. The general data structures given here are not the only ones that will work; any data structures that can provide the above mappings will be sufficient. However, the data structures presented here have been shown to allow the debugger to easily extract information. The data structures are based on the control-flow graphs used by the compiler. The nodes in the control-flow graph can be represented by data-flow graphs or by operation lists. The general structures given here can be modified based on the structure of the synthesizing compiler being used. Section 6.3 shows how these general structures were extended to create the debug database for Sea Cucumber.

Of all the mappings in the hardware line number table, mapping L3:OrigOp-FinalOp is typically the most complex. For this reason, the general data structures are based on making this mapping efficient and all other mappings are added to it. Correlating the original and final control-flow can be done in a number of ways. In work done in the field of debugging optimized programs, researchers have advocated storing the original graphs and the final graphs and correlating them during debug using graph matching techniques [17]. Others have advocated making the correlation using only annotations made to the graph during optimizations [57]. Though both of these methods work for simple optimizations, neither is convenient when predication and block merging is used. Because the final control-flow is much different than the original, it is difficult to either correlate the original graph with the final graph directly, or annotate the graph enough to allow the correlation. However, combining these two methods allows the correlation to take place. This method also has the benefit of providing appropriate places to insert the other mappings.

The general data structure for making these mappings contains two control-flow graphs: the Original Graph and the Final Graph. The Original Graph reflects the operation ordering given in the source code and is the starting point of the compilation flow. The Final Graph represents the optimized and scheduled application and is the end point of compilation and the starting point of synthesis. In order to make the mapping between original operations and final operations, all operations in the Original Graph that are derived directly from the source code are given a unique operation identification number

(ID). These IDs are then propagated during optimizations in such a way that it is possible to identify which of the final optimizations is accomplishing which initial operation. The way these IDs are propagated depends on the type of optimizations done. For example, if a constant multiply is split into multiple shift and add operations, then each of these new instructions would be annotated with the original operation ID. For reference, the rules used in Sea Cucumber are explained in Section 6.3.2. Figure 6.4 shows the original and final graphs for the example in Figure 5.6; the operation IDs are shown in parentheses on the left of the operation. How this information is interpreted depends on the mode in which the debugger is operating. Chapters 8 and 9 explain how these annotated graphs are used to extract the information necessary to observe and control the execution of the circuit for each of the operating modes.

Mapping L2:Op-InitSched is contained in the original control-flow graph, and these two graphs can also be used to efficiently store three of the other mappings: L1:Line-Op, L4:Op-FinalSched, and L5:Op-Pred. Mapping source lines to original operations (L1:Line-Op) can be done either of two ways. The first is to store the line number directly in the original graph data structures. In this case, each operation derived directly from a source line is annotated with the line number from which it was derived. The second option is to create a table which maps operations to line numbers by the unique IDs. An example of such a table for the graphs in Figure 6.4 is shown in Figure 6.5.

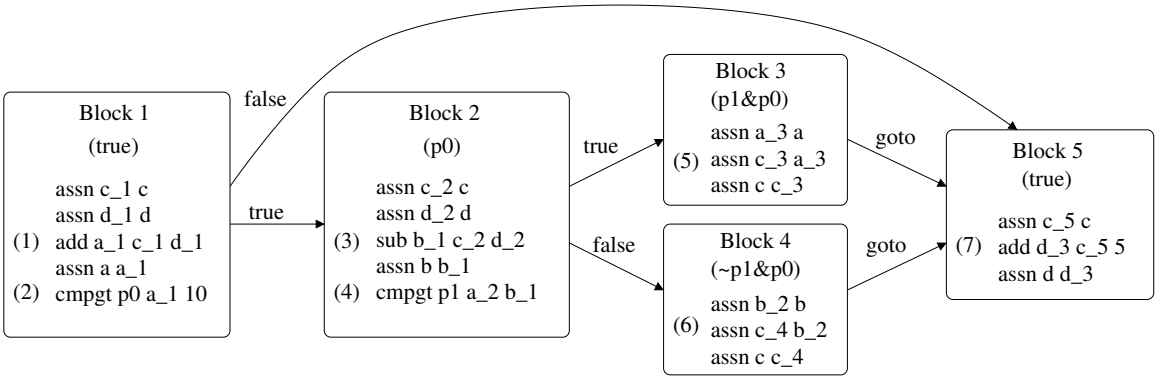
The mappings from operations to schedule (L4:Op-FinalSched) and operations to predicate equations (L5:Op-Pred) are done by simply annotating the operations in the final graph with this information. The final graph in Figure 6.4(c) is annotated with this information. In addition, if the predicate equations for the original instructions are desired, this information can be stored in the original graph. Each operation can be annotated with this information, or, as shown in Figure 6.4(b) each basic block can store this information.[4]

The last two mappings, L6:Sched-State and L7-BPU, are very dependent on the synthesis techniques used to map the application to hardware. For this reason their contents

[4]Because the original graph consists of basic blocks, each operation in the block will have the same predicate equation.

```
39 a = c + d
40
41 if ( a > 10 ) {
42   b = c - d;
43
44   if ( a > b )
45     c = a;
46   else
47     c = b;
48 }
49 d = c + 5;
```

(a) Source Code



(b) Original Graph

0	(1)	add a_1 c d	(true)
0	(3)	sub b_1 c d	(p0)
1	(2)	cmpgt p0 a_1 10	(true)
1		assn a a_1	(true)
1	(4)	cmpgt p1 a_1 b_1	(p0)
2	(5,6)	mux c_6 a_1 b_2 <~p1&p0>	(p0)
2		assn b b_1	(p0)
3		mux c_7 c_6 c <p0>	(true)
3		assn c c_6	(p0)
4	(7)	add d_3 c_7 5	(true)
5		assn d d_3	(true)

(c) Final Graph

Figure 6.4: Original and Final Graphs for Example in Figure 5.6

ID	Source Line
1	39
2	41
3	42
4	44
5	45
6	47
7	49

Figure 6.5: Source Line Mappings for the Example in Figure 5.6

will vary widely from synthesizer to synthesizer. This information could be stored in the two control-flow graphs, or separate tables could be created.

6.2.2 Accounting for Data-Flow: The Hardware Variable Table

Just as with the software variable table, the variable table in the debug database allows the debugger to find the storage location for variables, enabling the reading and writing of variable values. However, rather than mapping to a location in memory, or, in the case of Java, to a slot in the local variable array, the hardware variable table must map the variable to a location (circuit element) in the hardware. The debug database also needs to know to which type of circuit element the variable is mapped (i.e. register, memory, wire, etc.). This is important because the type of storage element will determine how the debugger treats the variable. For example, if the variable is not mapped to a state element (for example, to a wire rather than a register), then it is possible that the debugger may not be able to read or set its state directly.

In contrast to the software variable table, the hardware variable table may need to contain information about variables which are not found explicitly in the source code. This could include information about predicates and state variables in the hardware. The state of these variables may not be accessible to the user of the debugger, but this additional information may be needed to properly extract the important state from the executing circuit. The exact information required to be in the hardware variable table will depend on the compilation and synthesis scheme used.

Static Single Assignment Introducing static single assignment has a large impact on the data-flow of a program. Instead of having a single storage location for a variable, each assignment to the variable creates a new version of that variable (an SSA variable). This means that a single variable in the source code will be mapped to multiple storage locations in the final circuit, each being valid for different segments of the source code. Due to the presence of multiple locations for a variable, the debug database must be able to account for each version of a variable and where it is valid in the source code. This accounting is complicated by optimizations which can delete, add and merge variables. The debug database must be able to account for all modifications made to an SSA variable.

Hyperblock Formation Hyperblock formation also has some small effects on the data-flow of an application. When blocks are merged, it is sometimes necessary to add new SSA variables and muxing operations to be able to select the correct version of a variable for use in the newly merged block. The additional versions of variables must be accounted for in the debug database.

Incremental Mappings in the Hardware Variable Table

Just as with the hardware line number table, the hardware variable table provides a set of incremental mappings which can be used by the debugger to produce the full set of required mappings. The incremental mappings provided in the hardware variable table are discussed next. Just as with the mappings in the line number table, this work will refer to these mappings by their number and mnemonic.

V1. **Source Variable $\longleftrightarrow$ SSA Variable (SourceVar-SSA).** This mapping provides information about how the source variables in the original code are mapped to SSA variables. The forward mapping provides a list of all SSA variables derived from the specified variable. The reverse mapping gives the original variable from which each SSA variable was derived.

V2. **Initial SSA Variable $\longleftrightarrow$ Final SSA Variable (InitSSA-FinalSSA).** The introduction of static single assignment can lead to redundancy of the SSA variables. During

optimizations, it is likely that these redundant SSA variables will be deleted and/or merged. Block merging on the other hand can lead to the addition of new SSA variables. This mapping provides information about how the data-flow of the application is changed during the addition, deletion or merging of SSA variables. The forward mapping is used to map an initial SSA variable to the SSA variable which holds its value in the final circuit. The reverse mapping reports all the SSA variables whose value is stored in a final SSA variable.

V3. **SSA Variable $\longleftrightarrow$ Operation/Instruction (SSA-Op).** This two way mapping associates SSA variables with the operations in which they are assigned or read. The forward mapping is a list of all instructions in which the SSA variable is read and the one operation in which it is assigned a value. The reverse mapping returns all the SSA variables read and written in the specified operation. If desired, this mapping can be broken into two separate mappings, one which maps the reads of variables and one which maps the writes to variables.

V4. **Variables $\longrightarrow$ Circuit Element (Var-Circuit).** This mapping is used to map all internal variables to circuit elements in the final hardware. This includes mappings for the SSA variables, predicates and state variables, when necessary.

V5. **SSA Variable $\longrightarrow$ Hardware Width (SSA-Width).** This mapping provides the final hardware width of each variable. This information is necessary for the debugger to correctly display the values of signed variables during hardware debug.

General Data Structures Used to Provide Incremental Mappings

This section will discuss a set of possible data structures which can provide the mappings discussed above. As with the data structures discussed for the hardware line number table, the data structures used here are not the only ones that can provide these mappings, but the mappings presented here mesh well with the data structures used in the line number table. Some of the mappings in the hardware variable table can be provided by extending the data structures used in the line number table, others require additional tables to be added to the data structures.

The first mapping, V1:SourceVar-SSA, is a trivial, but necessary, mapping. It is best handled by allowing the data structure used to represent an SSA variable to keep track of the variable from which it is derived. This can be done explicitly or through the use of naming conventions (for example, where a_1 would represent version 1 of the variable a). Using this method means that the reverse mapping is quick and efficient. The forward mapping, however, would require a search through all SSA variables in order to find all instances derived from a particular variable. If a faster mapping is required, then a list of SSA variables can be formed for each variable.

Mapping V2:InitSSA-FinalSSA can be stored in a simple table. The original SSA variable is used as the lookup into the table and the final SSA variable is returned. If the key is not found in the hashtable, then it means that the variable was unchanged during optimizations. It is also possible to use this table to record SSA variables which were removed and why they were removed. This is accomplished by storing a special value for removed SSA variables which contains this information. The rules for building this table will depend on the types of optimizations used. For an example of the rules used by Sea Cucumber see Section 6.3.2.

Mapping SSA variables to operations (V3:SSA-Op) is best done by extending the data structures used in the line number table, to ensure that the DFG or data structure representing operations in the operation list contains information about the variables used by the operation. This information is not actually needed by the line number table, however, when present in the variable table, it can be used to extract the information about which variables are used in which operations. Again, this creates a very fast reverse mapping, but requires a search to provide the forward mapping.

The final two mappings, V4:Var-Circuit and V5:SSA-Width, can be provided in one of two ways. Either separate tables can be used to do this mapping, or the data structure representing variables in the final graph can have this information added to it.

6.3 Sea Cucumber Hardware Debug Database

The Sea Cucumber hardware debug database extends the general data structures described in this chapter. Since SC allows multiple independent threads, the debug database

contains a separate set of mappings for each thread. These mappings are built in several stages during the compilation and synthesis process:

Stage 1 **Original Graph Creation.** The original graph is created immediately after bytecode parsing and CFG/DFG formation. For the most part, this is a straightforward process. However, since the DFGs created by the bytecode parser do not maintain original instruction ordering within basic blocks (see Section 5.3.2), this ordering must be determined in order to create the original graph. The ordering is determined by sorting the operations based on the source line number from which the operations were derived. Sorting by line numbers means that there is no way to ensure that operations on the same line are in the correct order. However, as this information is only used in Source Step Mode, which is based on line number boundaries, it is not imperative that these operations be in the correct order. There is also one case in which sorting by line numbers does not work at all: the increment instruction in `for` loop headers. This arises because the increment instruction is actually put at the bottom of the loop body, which means that although it has the smallest line number in the loop, it must be placed at the end of the loop body. Therefore, these types of instructions must be detected and handled specially. Another fix would be to modify Soot to provide the bytecode offsets for the original operations. Then, the basic blocks could simply be sorted by the bytecode offset.

Stage 2 **Annotations During Optimizations.** During the optimization stage, Sea Cucumber keeps information about how the graphs were optimized. This information is then used in the creation of the final debug database. The annotations used during optimizations are discussed in Section 6.3.2.

Stage 3 **Final Graph Creation/Debug Database Formation.** After the compilation process is complete, the majority of the debug database is created. The final graph information is extracted from the internal SC data structures and added to the debug database. The annotations made during optimizations are also used at this stage to create the auxiliary data structures.

Stage 4 **Final Hardware Mappings.** During the creation of the generic circuit object, all circuit elements which represent SSA variables or state variables in the application are annotated with the name of the variable that they represent. When this object is converted to a JHDL circuit, the conversion engine reports the final hardware names of these variables to the debug database.

Details about what is done in each of these stages is given in the following sections. They will also discuss the contents and structure of the SC debug database, as well as discuss the issues arising from the optimizations used by Sea Cucumber.

6.3.1 Structure of the Debug Database

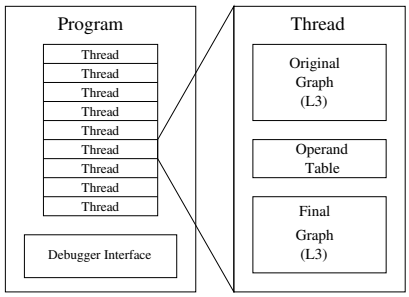
The structure of the debug database is based on the general data structures described earlier. The structure of the original and final graphs in the database parallels the hierarchical data structures used by the Sea Cucumber compiler. This structure is shown in Figure 6.6. The figure shows the main data structures as well as mappings to which each of the pieces contributes. The figure shows only the core data structures; other data structures are used to reduce the time required to make certain mappings, however, this information is all extracted from the main data structures. The contents and their significance for each of the pieces of the SC debug database are discussed in the following sections.

Program Level

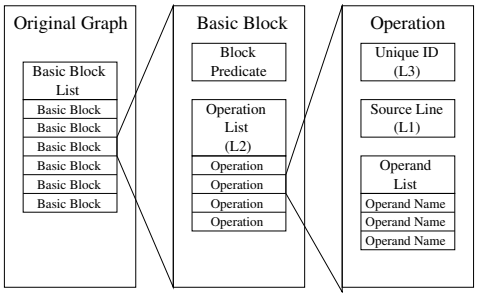
The top level of hierarchy in the debug database is called the `Program` level. Because the threads are treated independently in Sea Cucumber, the `Program` level simply contains a list of all the threads in the application. This is also the level at which the debugger first interacts with the debug database.

Thread Level

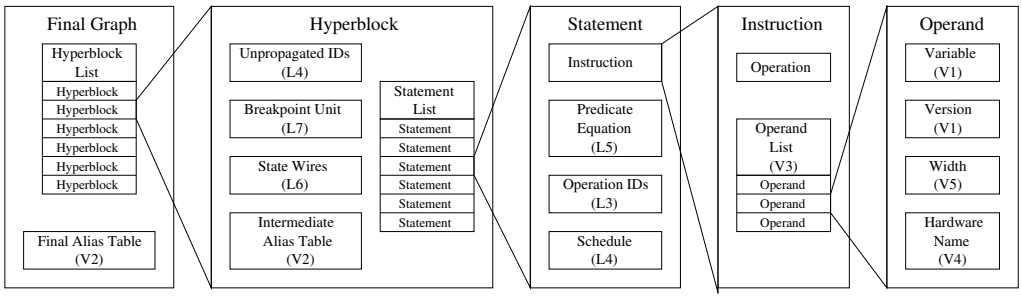
Because of the relative independence of threads, most of the debugger's interaction with the debug database is done at the `Thread` level. Each thread in the debug



(a) Program and Thread Levels



(b) Original Graph



(c) Final Graph

Figure 6.6: Structure of the Debug Database

database contains the original and final graphs, as well as the Operand Table discussed below. The other data structures described in this chapter are stored in the level of hierarchy in which they make the most sense.

Operand Table The Operand Table is used to map operand names to the Operand data structures. This table is added to increase the speed at which the debug database can make some of the incremental mappings.

Original Graph Level

The `Original Graph` Level is the top level of hierarchy for the graph which reflects the original control- and data-flow of the application. It contains a list of all the basic blocks generated by the bytecode parser. All information about operations in the basic blocks are kept in the `Basic Block` level.

Basic Block Level

The `Basic Block` Level stores information about a single basic block in the original CFG for the thread. This level contains a list of the original operations in the order in which they would execute in the original sequential source. The basic block also contains a `Block Predicate`. The `Block Predicate` gives the predicate equation for all operations in that block.

Each `Basic Block` also contains a list of all in and out edges and conditions by which they are taken. This information is needed to have a complete graph.

Operation Level

The `Operation` Level of the graph contains information about a single operation in the original control-flow. It contains a list of the names of the operands used in the operation and whether they are written or read. It also contains the unique ID assigned to the instruction at the time the original graph was built. The final piece of information contained in the `Operation` level is the line number in the original source code from which this operation was derived.

Final Graph

The `Final Graph` Level encapsulates all the information about the application after in has been optimized. The graph structures in this level mimic the structures used by Sea Cucumber and consists of a list of hyperblocks that each contain a list of statements. This level also contains the final alias table.

Final Alias Table The final alias table is a Hashtable which maps an SSA variable (operand) to the final operand which holds its value after optimizations. A void mapping means that the operand was not changed and that it stores its own value. Those operands which were removed for any reason return the string REMOVED. In this way, the debugger can get the final storage location for an SSA variable or determine that it has been removed.

Hyperblock Level

The `Hyperblock` Level contains information about one hyperblock in the final graph. The main piece of information is a list of the statements contained in the hyperblock. It also contains other information which effects the entire hyperblock. These pieces of information are discussed below.

Unpropagated IDs This is a list of IDs from operations which were removed during the Propagate Alias optimization, but for which the operation IDs could not be properly propagated. This happens when the IDs would need to be propagated outside of the current hyperblock. Along with the operation IDs, the name of the primal which was aliased by the removed operation is included. For more information, see the description of the Propagate Alias optimization in Section 6.3.2.

Breakpoint Unit This data structure contains information about how the breakpoint unit for each hyperblock is connected to the signals in the rest of the hyperblock. This data structure is a Java class and has methods associated with it that can create the programming stream for the database based on a schedule and predicate equation. For a detailed look at the breakpoint circuitry inserted by Sea Cucumber, see Chapter 7.

State Wires Each hyperblock contains a list of the hardware names for each of the one-hot [58] state bits in its control unit. This allows the debugger to determine the circuit elements which represent the state of the executing circuit.

Intermediate Alias Table During compilation each hyperblock contains an Intermediate Alias Table. These Alias Tables are combined after compilation into the Final Alias Table found in the `Thread` level. The intermediate tables are needed in the hyperblocks because of the presence of primal variables. When merging blocks, reads from primal variables may be changed into non-primal reads; the alias table must be modified to reflect this change. If a single Alias Table is used at the `Thread` Level, then it is difficult to determine what needs to be changed in the table. This is discussed in some detail in Section 6.3.2 when looking at the Merge Serial optimization.

Statement Level

The `Statement` Level contains information about a single statement. A `Statement` consists of an `Instruction` as well as its Predicate Equation, operation IDs and Schedule.

Predicate Equation The predicate equation is stored in sum of products form and is simply a two dimensional array of the single predicates and whether or not they are inverted. The predicates in the inner arrays are `and`'ed together and these results are `or`'ed together. This information allows the debugger to determine the conditions under which the results of this operation will be committed.

Operation IDs This is an array which has the IDs for all operations which are performed by the current statement. For information on how this list of IDs is formed, see Section 6.3.2.

Schedule This gives the logical schedule for the statement. The schedule is simply the number of the state during which this instruction executes. The complete schedule for the

instruction also includes the hyperblock in which the statement resides, but this information is inferred from the hierarchy of the data structures, so does not need to be stored here.

Instruction Level

The `Instruction` Level contains information about the specific instruction that is part of a statement. The instruction has information about the type of operation as well as a list of all the operands used in the instruction.

Operand Level

The `Operand` Level contains information about individual operands. Operands can be either SSA variables or predicates generated by comparison instructions. For SSA variables, it contains the source variable name, as well as its version number. Primal variables are given a version number of -1 to differentiate them from the versioned variables. After synthesis is completed, the `Operand` also stores the final storage width of an SSA variable, as well as the hardware name. The hardware name is the name given to the circuit element which represents that operand in the JHDL circuit.

6.3.2 Debugging in the Presence of Optimizations

To provide a better understanding of how the optimizations affect the creation of the debug database, the next paragraphs will describe the optimizations used by SC and how they affect (or don't affect) the creation of the database. The optimizations can be organized into three major groups: size reduction, operation simplification and parallelism enhancement. Table 6.3.2 lists all the optimizations used in Sea Cucumber.

Remove Unused Code

Remove Unused Code seeks to remove the unnecessary computations from a hyperblock. A pessimistic approach is used, in which all operations must prove their use in the hyperblock; any operation or predicate which does not directly or indirectly affect an update to a thread register or is not necessary for control-flow is removed from the hyperblock.

Table 6.1: SC Optimizations

Size Reduction	Operation Simplification	Increased Parallelism
Remove Unused Code	Simplify Stack Variables	Unroll Loops
Common Subexpression Elimination	Propagate Alias	Merge Parallel
Move Loop Invariants	Constant Math	Merge Serial
Bit Width Analysis	Constant Mux	Instruction Scheduling
Remove Empty Blocks	Identity Reduction	

Since code is only removed if its result is unused, it is important that the resulting operand of the operation is reported to the Alias Table as having been removed. This allows the debugger to report this variable as being unused. Removal of unused code has the consequence of not allowing a breakpoint to be set at the originating line of source code.

Instructions IDs are not propagated during this optimization. Since the IDs are always propagated for useful code, the lack of an instruction in the final graph is an indication to the debugger that the operation was removed.

Common Subexpression Elimination

Common Subexpression Elimination (CSE) looks for redundant operations in a hyperblock and removes them. It does this by creating a new SSA variable and operation. The result of the redundant operation is written to the new SSA variable and the original computations are replaced with assignments from the new SSA variable. The new assignments preserve the predication and operation IDs of the original operations.

Move Loop Invariants

The Move Loop Invariants optimizations reduces size requirements by removing code from a loop. This allows the execution to be done once, instead of many times in the loop. This optimization is not supported by the SC Debugger as it moves the code outside of its original block. instruction is marked as loop invariant in the debug database. When in clock step mode, the loop invariant code will execute only once, before or after the body of the loop. Thus, if the user sets a the might be

Bit Width Analysis

Bit width analysis has little effect on the presence of instructions but reduces the number of bits or ensures the number of bits for a given operation are correct. This is accomplished by Sea Cucumber using the method described in [59]. Functions describe the bit width in forward analysis and backward analysis. Forward and backward width analysis are completed separately until the values of the widths stop changing.

After the bit widths for all variables have been determined, they are reported to the debug database. Because all variables in SC are signed, this information is needed by the debugger in order to correctly display the values for each of the variables.

Simplify Stack Variables

The Simplify Stack Variables optimization was added to facilitate the creation of the debug database. Specifically, it is run to clean up unneeded stack variables before the Original Graph is built. In the graphs created during bytecode parsing, all instructions, except assignments, consist of two operations. The first computes the result and stores it in a stack variable, the second assigns the stack variable to the actual variable. For example, the instruction:

```
a = b + c
```

would be broken up into the following two operations:

```
add %int_1  b_1  c_1
assn  a_1 %int_1
```

where $\%int\_1$ is a stack variable. After the Simplify Stack Variables optimization is run, the two operations are reduced to a single operation. In this example, this operation would be:

```
add  a_1  b_1  c_1
```

The Propagate Alias optimization described next takes care of this clean up when debugging is not turned on. However, it can also change the control- and data-flow of the program, so it is not able to be run before creation of the Original Graph.

Propagate Alias

The goal of Propagate Alias is to remove operand aliases in the SC intermediate code. Operand aliases obscure the true nature of a value and do not correlate well with the original source code. Operand aliases are generated three different ways: 1) as stack variables from the Java bytecode, 2) by assignment and 3) as aliases made during hyperblock formation. As mentioned above, Propagate Alias removes the stack variables if they are present. Similarly, assignment operations which result in merely an alias being generated are removed and the alias name is replaced with the original source name. Finally, when two basic blocks that read the same thread variable are merged, the internal SSA name for the variable is different, despite the fact that it comes from the same source. The different aliases are replaced with the source operand name. In this way, aliases are removed in order to better understand where a value is generated.

This optimization raises two important issues. The first is the replacement of aliases with their actual sources. These changes need to be recorded in the Alias Table. This is done at the same time that the alias is replaced in an instruction. The other issue arises when the optimization removes the original assign instruction that created the alias. If this instruction is associated with an operation ID. then that information needs to be propagated to the correct instruction.

Updating the Intermediate Alias Table The Propagate Alias optimization updates the Intermediate Alias Table contained in the current block. When an alias is replaced by the original source, the information is added to the alias table. The key for the entry is the alias, and the value is the original source.

Propagation of Instruction IDs When removing the original assignment, Propagate Alias must correctly propagate the instruction IDs of the assignment. The instruction IDs for the removed instruction are propagated to the instruction which assigns the variable which was being assigned from in the removed operation. For example, take the following code segment:

```
add a_4 b_2 c_2    (ID: 1)
assn d_1 a_4       (ID: 2)
sub b_3 d_1 e_2    (ID: 3)
```

Running the propagate alias optimization would result in:

```
add a_4 b_2 c_2    (ID: 1,2)
sub b_3 a_4 e_2    (ID: 3)
```

The instruction IDs from the `assn` operation are propagated back to the line where `a_4` was originally assigned. This is done because, in effect, `d_1` is now assigned at the same time as `a_4`, as the value of `d_1` is now stored in the SSA variable `a_4`.

If the instruction which assigned the source variable is not in the current block (i.e., the variable is an alias of a primal), then the instruction IDs and the source primal are recorded in the Unpropagated ID Table for the block. This information is then propagated when appropriate during the merging operations. Any variable left in the Unpropagated ID Table after optimizations is considered to run just before the block starts execution.

Constant Optimizations and Identity Reduction

Constant Math and Constant Mux look for operations where either of the inputs are constant or the select line for the mux is constant. Identity Reduction operates in a similar way, except that it searches for mathematical identities to reduce. Table 6.3.2 shows all of the identities that are reduced. All of the operations replace a more complex instruction with a simple assignment of the value. Since these optimizations are not merging or deleting operations, they must simply ensure that the Operation IDs for the replaced instruction is propagated to the new instruction.

Unroll Loops

Unroll Loops attempts to increase parallelism by increasing the number of instructions available for scheduling. As mentioned previously, loop unrolling is not supported by the Sea Cucumber Debugger.

Table 6.2: Identity Reduction

Arithmetic reduction

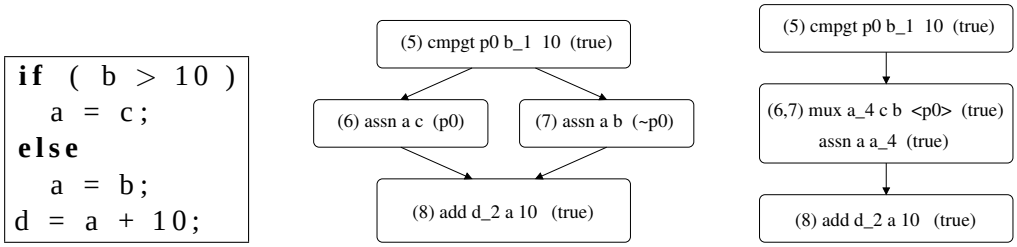
x + 0	=	x			
x - 0	=	x	0 - x	=	-x
x $\times$ 1	=	x	x $\times$ 2^i	=	x << i
x $\times$ 0	=	0	x $\times$ -1	=	-x
x / 1	=	x	x / 2^i	=	x >> i
x >> 0	=	x	x << 0	=	x

Merge Parallel

Merge Parallel and Merge Serial are steps within the optimization path which create hyperblocks. Initially, the data-flow graphs are made up of operations, which are grouped into basic blocks. After if-conversion and predication, these basic blocks can be combined into hyperblocks. Basic blocks and hyperblocks can be merged across forward control-flow edges either serially or in parallel. Merging reduces the number of blocks and makes more operations available for concurrent scheduling.

The Merge Parallel optimization merges blocks which have the same source and sink blocks. This type of structure in the control-flow graph typically arises from `if-then-else` statements. In a parallel merge, the only conflict that can arise is a conflict on primal writes. If both blocks write to the same primal, then a `mux` operation must be inserted to determine which value will actually be written to the primal. The mux selection is based on the predicate equations of the original operations, and the output of the `mux` operation is then written to the primal variable in question. If the primal writes have operation IDs associated with them, then they are propagated to the `mux` operation that replaces them.

Figure 6.7 shows an example of merging two parallel blocks when there is an output conflict. Notice that both blocks write to the primal variable `a`. On merging, these assignments are replaced by a `mux` operation and their operation IDs, 6 and 7, are propagated to the new operation. The result of the mux operation `a_6` is then written to the primal `a`.



(a) Source Code (b) Original Basic Blocks (c) Merged Block

Figure 6.7: Example of a Parallel Merge

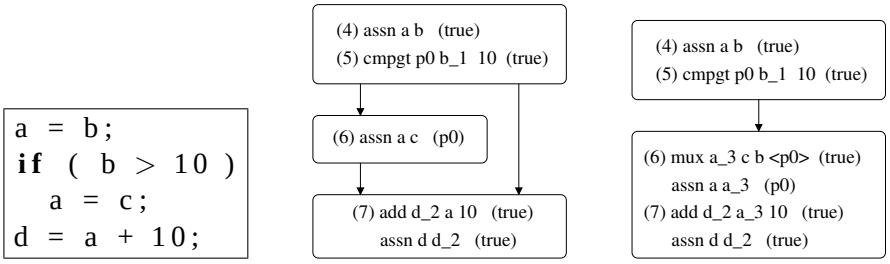
In addition, the debug database also records which basic blocks were merged into each of the final hyperblocks. This allows the database to better correlate the original control-flow of the program with the final control-flow of the circuit.

Merging Alias Tables After the statement lists are merged, Merge Parallel must also merge the intermediate alias tables for the two blocks into a single block. With parallel merges, the tables can be merged directly without modification. This is done by simply adding all key-value pairs from one table into the other table.

Merge Serial

The Merge Serial operation merges blocks which have a single forward in-edge into the block from which the in-edge originates. Just as with a parallel merge, the serial merge can also have output conflicts. The Merge Serial optimization handles these types of conflicts in the same way as a parallel merge: the primal writes are replaced with a `mux` operation and a new primal write.

The Merge Serial optimization can also create another type of conflict, the output-to-input conflict. This conflict arises when the source block writes to a primal variable and the sink block reads from the same primal. Since Sea Cucumber does not allow reads from primals after the block has written to that primal, the sink block must read from the intermediate variable in the source block instead. However, since the source block may



(a) Source Code

(b) Original Basic Blocks

(c) Merged Block

Figure 6.8: Example of a Serial Merge

not write the primal, depending on the predicate equation of the write, the sink block may need to read from the primal instead of the source's value. This means that a `mux` must be added to determine which value is used. All reads from that primal are then replaced with the output of the `mux` operation.

An example of a Serial Merge with this type of conflict is shown in Figure 6.8. This type of conflict requires Merge Serial to modify some of the annotations used to create the debug database. These are discussed in the following paragraphs.

Merging Alias Tables If no output-to-input conflicts occur, then the alias tables can be merged directly just as with the Merge Parallel optimization. However, when an output-to-input conflict does happen, then it is possible that one or more entries in the sink block's Alias Table will need to be changed. Any reference to a primal with an output-to-input conflict must be changed to refer to the output of the new `mux` operation. After these changes are made, the two Alias Tables can be merged.

Unpropagated IDs and Output-to-Input Conflicts If the primal variable involved in an output-to-input conflict is found in the Unpropagated IDs table, then the operation IDs associated with it must be propagated to the new `mux` operation. The operation IDs were originally left unpropagated because the primal reads had been removed by the Propagate

Alias optimization. However, now that the `mux` operation is available, the assignment to the variable can be associated with it.

Whether or not conflicts existed, the two Unpropagated IDs Tables are merged, so that the new block contains the Unpropagated IDs from both blocks.

Instruction Scheduling

Sea Cucumber has the ability to allow for different schedulers to be used during compilation. The scheduler currently used when debugging is turned on uses an ASAP scheduling algorithm, and each operation is assumed to take a single clock cycle to complete. Though instruction scheduling has a big impact on how the debugger operates, the debug database must simply store the schedule for each instruction. This information is then interpreted by the debugger.

6.4 Conclusions

Just as with software debuggers, the hardware debugger requires the compiler to keep track of information on how the application was mapped to the final architecture. However, the hardware debug database must provide more mappings than the software symbol table. For example, the software line number table simply maps lines of source code to instructions in the object code. The hardware debug database makes a similar mapping in its line number table (incremental mapping L1:Line-Op), but must also make six other mappings. The main reasons for the increase in the number of required mappings lies in the fact that there is no fixed architecture to which the synthesizing compiler is compiling code and the fact that many optimizations are performed which greatly affect the control- and data-flow of the program.

Despite the increase in complexity for creating the hardware debug database, the necessary mappings can be made using a relatively small number of simple incremental mappings. The use of incremental mappings makes it easier for the synthesizing compiler to keep track of the necessary information for debugging the application at the source level.

Chapter 7

Debug Hardware

This chapter will discuss the extra hardware that the synthesizer must add to the final circuit to enable certain features in the debugger. In general, there are three main categories of hardware that must be added. The first is breakpoint hardware; the second is buffering hardware; and the third is checkpointing/rollback hardware. It is also possible that hardware will need to be added to allow clock control and state readback and setting; this was discussed in Chapter 4.

The next sections will discuss, in general terms, and with an example from the Sea Cucumber compiler, the issues in providing these three types of hardware. The exact nature of the additional hardware is dependent on how the synthesizer generates circuitry from source code.

7.1 Breakpoint Hardware

The breakpoint hardware (referred to as the breakpoint unit or BPU) must be able to identify when an operation is executing and stop circuit execution at this point. The BPU determines when to stop execution by examining relevant predicate values and control state of the executing circuit. The actual design of the BPU will depend on how the synthesizer organizes the circuit control and the computation of predicates.

Since SC uses distributed state machines for control, the synthesizer adds multiple breakpoint units, one for each hyperblock in the circuit. The inputs to the BPU are the state bits from the hyperblock control, and the predicates computed in the hyperblock datapath. The BPU circuitry is shown in Figure 7.1.

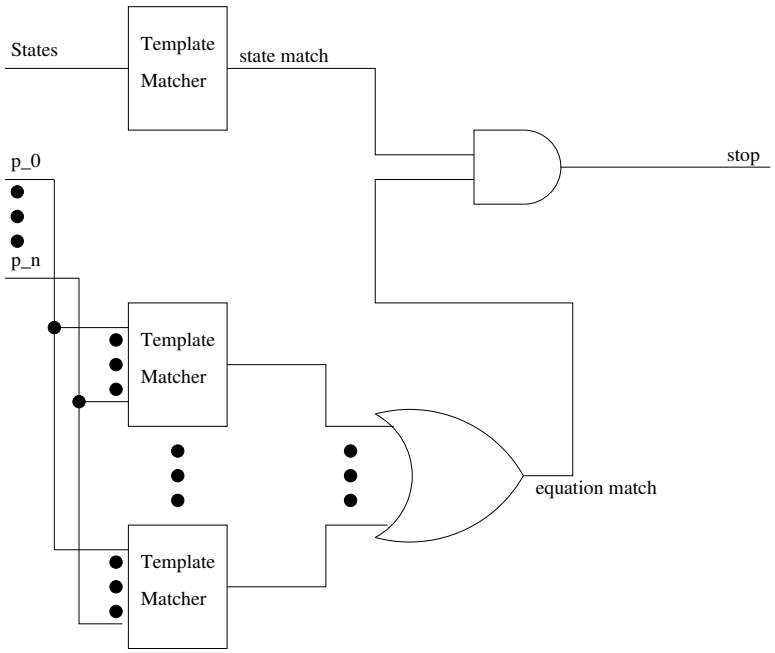


Figure 7.1: Breakpoint Unit Circuitry Used by Sea Cucumber

The circuit is made up of two parts: a programmable template matcher to match the correct cycle, and programmable circuitry to compute the results of the predicate equations, for the instructions on which the breakpoints will be set. The circuitry computes the predicate equation in sum of products form. The products are computed using programmable template matchers. These results are then *or*'ed together to produce the result of the equation. The programmable template matchers take all the predicates generated in the hyperblock as input. The number of template matchers is determined by the predicate equation with the maximum number of terms. This allows the circuity to compute all of the predicate equations in the hyperblock. If the current cycle template matcher and the predicate equation circuitry both provide positive results, then the `stop` signal is asserted, which will stop the circuit execution. In order to keep circuit frequencies high, the stop signal can be registered. This will make the circuit stop one cycle later than expected. However, in source step mode this does not matter (see Section 9.2.2), and in clock step mode, the debugger can use the method which will be described in Section 8.2.2 to make the breakpoint stop on the correct cycle.

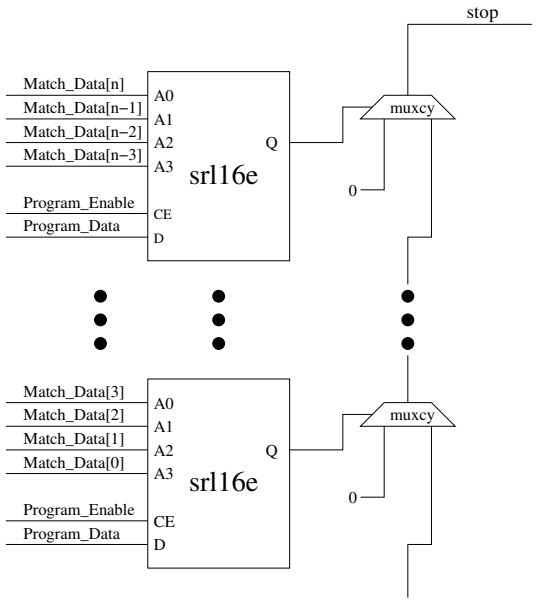


Figure 7.2: Programmable Template Matcher Circuitry

The `stop` signal generated by the BPU is used both to stop the circuitry and to notify the debugger that a breakpoint was reached. Notifying the debugger can be done by any convenient means. On the Slaac-1V board, the signal is connected to the register interface (see Section 4.2), and the debugger polls the hardware to determine when a breakpoint was reached. There are other methods which would work equally well for notifying the debugger when a breakpoint is reached. For example, it would also be possible to wire the `stop` signal to an interrupt line, if such a feature is supported on the target platform..

BPU template matching circuitry can be programmed at run-time. The structure of the programmable template matchers is shown in Figure 7.2. Each `srl16e` in the figure is a "16-bit shift register look-up-table (LUT) with clock enable"[44]. Each `srl16e` can hold 16 bits of data. When enable is asserted, each bit is shifted one location. The bit shown on the output is based on the value on the address ($A0 - A3$). When the `srl16e` is not enabled, it behaves like a normal lookup table (LUT).

Each `srl16e` in the template matcher matches a 4-bit value. The 4-bit template matchers are hooked together by the Virtex carry logic. The carry logic is configured to

and together the outputs of the 4-bit template matchers. The matcher is programmed by asserting the `Program_Enable` signal. The new programming data is then shifted in one bit per clock cycle on `Program_Data`.

Once the design of the BPU has been determined, it must be determined how the unit will stop execution of the circuit. The breakpoint circuitry can stop the execution of the circuit in one of two ways, depending upon what support is available in the target hardware. The first approach simply stops the relevant clock when a breakpoint is reached. This requires the hardware to be able to stop the clock in a single clock cycle. The second approach uses control circuity to hold the circuit in its current state when a breakpoint is reached. This is called an architectural stop and has the consequence of only stopping execution of a single thread. All other threads will continue to execute to synchronization points. However, under the assumption that threads can run independently, this does not lead to unpredictable behavior.

The SC Debugger currently uses the architectural stop, as the test platform, the Slaac-1V board, does not support single cycle clock stopping. For the architectural stop, the `stop` signal (see Figure 7.1) is used to prevent all flip-flops in the current hyperblock from updating. This includes the state bits in the control unit, as well as all registers in the datapath.

7.2 Checkpointing/Rollback Hardware

The checkpointing/rollback hardware is used to return the circuit to a specific time in the execution. This hardware is used in both clock step and source step modes of the debugger. Before the hardware can be added, a decision must be made about where the checkpoints will be taken. In this work, we chose to make the checkpoints at the first cycle of execution of a hyperblock. There are several reasons for this decision:

1. Ease of implementation.
2. Original control-flow and final control-flow match up at the hyperblock boundary.
3. When used for setting of variables in source step mode, this ensures the coherency of the block's execution (see Section 9.2.5).

The rollback hardware consists of buffering and control circuitry that can force circuit execution to return to the last checkpoint. Because the checkpoints are made at the beginning of each hyperblock, the synthesizer adds hardware to allow the state machines controlling the execution of a hyperblock to return to the first cycle of execution.

Sea Cucumber-synthesized circuits use distributed control units. Rollback control hardware is simply created by adding an extra input to each hyperblock state machine, called the `rollback` signal. When the `rollback` signal is asserted, the state machine returns to the first state of execution, unless it is currently in the wait state, in which case it remains in the wait state. This ensures that only the active hyperblock in each thread will be affected by the rollback signal.

7.3 Buffering Hardware

In addition to storing state information that will be used to checkpoint/rollback a circuit, the previously described buffering hardware is also used to store past values of variables. This is necessary in source step mode when a value is still needed for Virtual Sequentialization (see Section 9.1), but has been overwritten by the circuit.

However, this additional functionality usually does not affect the size of the buffering hardware because the amount of data that must be stored for rollback is larger than that required to provide storage for previous variable values. In this case, since the checkpoints are made on hyperblock boundaries, the buffering hardware need only be added for resources which are reused/shared during the execution of a block. The use of SSA provides much of this buffering because a variable version is only assigned once.

In circuits generated by Sea Cucumber, the only shared resources which need to be buffered are the thread registers. They are buffered because they can be updated anytime during the execution of a hyperblock, but the old values may be needed after these updates. The thread registers are buffered during the first cycle of execution of a hyperblock, before any of the registers are overwritten. Simply buffering the registers is sufficient for watching variables, but in order to rollback circuit execution, the buffered state must be written back to the original registers. Both of these are accomplished by using a shadow register, as shown in Figure 7.3, for each thread register.

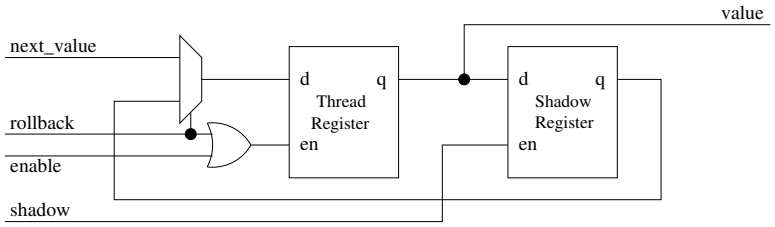


Figure 7.3: Buffering Hardware Added to the Thread Registers. This hardware is used both to facilitate the watching of variables, as well as to allow circuit execution to be rolled-back to the beginning of the current hyperblock.

Asserting the `shadow` signal causes the shadow register to buffer the current value of the thread register. This signal is asserted on the first cycle of execution for each hyperblock. Asserting the `rollback` signal will cause the shadowed value to be written back into the thread register.

7.4 Controlling the Debug Hardware From the Debugger

In order to use the debug circuitry, the debugger must have a way to communicate with it. There are two ways to provide this communication. The first is to use one of the state writing techniques described in Chapter 4. The second approach is to create a set of global control lines which can be used to control the additional circuitry. Each of these approaches is discussed below.

7.4.1 Control through Writing State of Device

In this approach, the debugger uses the ability to write the values of state elements on the device. The major advantage of this approach is that the added debug circuitry can be simplified somewhat. However, this approach also inherits whatever disadvantages the method of state setting brings with it, which include increased area and speed overhead for scan chain, and the inability to use the built-in set/reset capabilities on flip-flops if using bitstream manipulation on Xilinx Virtex devices.

If this approach is used, then each of the circuit additions can be simplified. In the case of the breakpoint unit, the `srl16e`'s in the template matchers can be replaced

with ROMs or RAMs, because the ability to serially shift in new data is no longer necessary; the memory elements are simply programmed by changing the contents using the state setting ability. The rewriting circuitry in the buffering hardware can also be left out, as the debugger can read the state of the shadow registers and then write the values to the thread registers through the state writing interface. And finally, the rollback hardware in the state machines can be left out completely, as the debugger can simply set the control unit to the desired state.

7.4.2 Control through the Use of Global Debug Signals

The second approach is to use global control lines for communicating with the additional circuitry. Unfortunately, the large fanout on these global lines can have a significant impact on circuit speed. However, the debugger, with one exception, only communicates with the debug circuitry while the circuit clock is stopped. This means that the debug signals can be set and allowed to settle before the circuit clock is advanced and the values are used. Thus, the timing of these lines can be ignored and does not impact the operational speed of the circuit. The one exception of communicating while the clock is stopped is polling the circuit to see if a breakpoint is reached. However, it is not necessary to find out immediately that a breakpoint was triggered, so again, we can ignore the timing of the signal which carries this information.

Because of the large number of debug units to be controlled, an addressing scheme is introduced. The hardware was developed such that only one component needs to be enabled at any given time. Each enable signal is given an address. When the address is presented on the global address line, the appropriate enable is asserted. Likewise, there is a global data line which is used to program the breakpoint units.

Another consequence of this approach, is that the debug-circuitry clock must be independent of the circuit clock, or there must be a way to stop the circuit from advancing while we clock the debug circuitry. This is accomplished by adding another global signal which, when asserted, causes all `stop` signals in the breakpoint units to assert. This effectively stops all thread circuitry from advancing.

7.5 Size and Speed Implications

Including debug hardware in the synthesized circuit adds extra overhead to the design. To help quantify this overhead, this section will provide size and speed overheads for three benchmarks. These include SC implementations of a cordic, the discrete cosine transform (DCT) and the tiny encryption algorithm (TEA) using 8 rounds. These applications were used as they are typical of applications which may be mapped to FPGAs because of their inherent parallelism and/or ability to be easily pipelined.

Table 7.1 gives the sizes (for both LUT and flip-flip usage) and maximum clock frequencies of each of these applications for five different configurations: no debug hardware added (none), all debug hardware added (full), only breakpoint hardware added (bpu), only rollback hardware added (rollback) and only buffering hardware added (buffer). The frequency (denoted as MHz in the table) values were obtained by running the circuits through the Xilinx place and route tools several times using different timing constraints. The maximum frequency obtained through this method is shown in the table. The table also gives the absolute and percent differences for each configuration compared to the circuit with no added debug circuitry.

As can be seen in the table, the overhead of adding debug circuitry is minimal, only about 2-9% for size and up to 10% for speed. However, different application characteristics can lead to larger overhead for each type of debug hardware. It is important that the end user understand what these characteristics are. The following sections will discuss this for each of the three debug units.

7.5.1 Breakpoint Unit

The size of the breakpoint unit is dependent on three factors: the number of states in a hyperblock, the number of predicates computed in a hyperblock and the maximum number of *or* terms found in a predicate equation in a hyperblock. In addition, there is also overhead involved for each registered variable in the application. This overhead comes from the need to gate the enable signal for each register in order to stop flip-flops from updating when a breakpoint is reached. This leads to a high overhead percentage when the variable widths are small.

Table 7.1: Circuit Sizes and Speeds

		Cordic			DCT			TEA		
		Size	Diff	%	Size	Diff	%	Size	Diff	%
None	LUTs	4762	-	-	3006	-	-	3378	-	-
	FFs	5753	-	-	4384	-	-	4507	-	-
	MHz	87.6	-	-	111	-	-	120.1	-	-
Full	LUTs	4967	205	4.3%	3179	173	5.8%	3683	305	9%
	FFs	5983	230	4.0%	4454	70	1.6%	4574	68	1.5%
	MHz	104.2	-16.6	-19.0%	100	11	9.9%	113.7	6.4	5.5%
BPU	LUTs	4891	129	2.7%	3116	110	3.7%	3581	203	6%
	FFs	5759	6	0.1%	4390	6	0.1%	4511	4	0.1%
	MHz	81.1	6.5	7.4%	108	3	2.7%	115.3	4.8	4%
Rollback	LUTs	4885	123	2.6%	3108	102	3.4%	3563	185	5.5%
	FFs	5977	224	3.9%	4448	64	1.5%	4571	64	1.4%
	MHz	101	-13.4	-15.3%	95.5	15.5	14%	107	13.1	10.9%
Buffer	LUTs	4763	1	0%	3007	1	0%	3379	1	0%
	FFs	5977	224	3.9%	4448	64	1.5%	4571	64	1.4%
	MHz	111.5	-23.9	-27.3%	112	-1	-1%	122.7	-2.6	-2.1%

7.5.2 Rollback Hardware

The overhead of the rollback circuitry is split into two parts: the overhead from the hardware used to reload values into the thread registers and the overhead from the circuitry used to return the control units to their start state. The sizes of each of these parts is determined by different factors. The hardware added to the thread registers naturally increases in size when the number of total bits in the thread registers increases. This leads to a large overhead when the amount of computation done in the hyperblocks is small.

On the other hand, the circuitry added to the control unit increases in size with the number of states present in the control unit. This translates to high overhead if there is a relatively small number of operations, on average, executing each clock cycle. However, this means that there is very little exploitable parallelism, and these types of applications do not see much, or any, of a performance advantage by being mapped to FPGAs.

7.5.3 Buffering Hardware

As can be seen in Table 7.1, the buffering hardware only adds overhead in flip-flop usage. An extra flip-flop is added to the design for each flip-flop in the thread registers. The overall overhead is small when the ratio of bits registered in the thread registers is small compared to the number of bits registered in the hyperblocks in a thread. This will be the case when the hyperblocks compute a large number of intermediate values which are not needed in other hyperblocks. This situation usually occurs for applications which do not have multiple small loops, which limit the potential size of hyperblocks.

7.6 Conclusions

While the basic functionality for hardware debugging can be provided by the target platform, the synthesizer must be able to add specific hardware to allow implementation of some of the debugger features. This hardware includes breakpointing hardware, rollback hardware and buffering hardware, and requires a modest amount of circuit overhead for typical circuits. The use of this circuitry by the debugger is discussed in the next chapters. The exact implementation of this hardware is dependent on the structure of the synthesized circuit. Sea Cucumber was modified to add this additional circuitry to the synthesized hardware when debugging support is enabled in the compiler. When debug support is turned off, this circuitry is not added, allowing the circuit to operate at full speed. Being able to add the debug circuitry while debugging the circuit and later removing it is a big advantage of using reconfigurable hardware.

Chapter 8

Clock Step Mode

As explained previously, the two possible operating modes of the hardware debugger are differentiated by the definition of a single step. In clock step mode, a single step is defined as advancing the hardware clock a single cycle. This mode provides *truthful* behavior in that it provides the user with a direct view of the state of the running circuit in the context of the source code. The clock step mode debugger does not attempt to interpret the state of the circuit, beyond that which is required to map the data back to the source code.

Providing a direct view of the execution of the circuit has distinct advantages and disadvantages, both to the creator and the user of the debugger. The main advantage for the user of clock step mode is that it provides the user with insight into how the synthesizer mapped the original code to the final optimized circuit. A disadvantage of this approach is that the view can be confusing to users who are accustomed to debugging code on computers which run the code sequentially. Clock step mode does not attempt to adhere to the flow of the original program, but rather provides the user with a view of the final flow of program execution.

This mode provides the advantage of having a simple paradigm for allowing control of the circuit. On the other hand, it poses interesting problems in providing intuitive feedback about the state of execution, which proceeds in parallel instead of sequentially.

8.1 General Issues for Clock Step Mode

In the process of providing the feature set of the debugger, the debugger uses the incremental mappings provided in the debug database to build up larger mappings. Since

many of these larger mappings are used to provide multiple functions in the debugger, this section will outline how these larger mappings are built from the incremental mappings. For reference, the incremental mappings are provided in Table 8.1. Many of the larger mappings provided here are used in both clock step mode and source step mode.

Table 8.1: Incremental Mappings Found in the Hardware Line Number and Variable Tables

L1:Line-Op	Source Line $\longleftrightarrow$ Original Operation
L2:Op-InitSched	Original Operation $\longleftrightarrow$ Initial Schedule
L3:OrigOp-FinalOp	Original Operation $\longleftrightarrow$ Final Operation.
L4:Op-FinalSched	Final Operation $\longleftrightarrow$ Schedule.
L5:Op-Pred	Operation $\longrightarrow$ Predicate Equation.
L6:Sched-State	Schedule $\longleftrightarrow$ Circuit State.
L7:Op-BPU	Operation $\longrightarrow$ Breakpoint Unit Programming Data.
V1:SourceVar-SSA	Source Variable $\longleftrightarrow$ SSA Variable
V2:InitSSA-FinalSSA	Initial SSA Variable $\longleftrightarrow$ Final SSA
V3:SSA-Op	SSA Variable $\longleftrightarrow$ Operation/Instruction
V4:Var-Circuit	Variables $\longrightarrow$ Circuit Element
V5:SSA-Width	SSA Variable $\longrightarrow$ Hardware Width

8.1.1 Determining Values of Variables

A key part of being able to operate the debugger is being able to extract the raw state of the circuit and convert it into useful information for the debugger. This includes finding the value of both the application (control circuitry, etc.) variables and the state variables controlling the execution flow of the program. This information must be recomputed each time the circuit is allowed to advance by either single stepping or running to a breakpoint. This is done by first extracting the state of the executing circuit using one of the methods described in Chapter 4. Mapping V4:Var-Circuit is then used to map the raw circuit data to information about the values of the variables in the application. The

debugger also uses Mapping V5:SSA-Width to ensure that the final width of the variable is taken into account when determining the variable value. This information is then stored globally in a global repository and used by the debugger to get the values of the SSA and state variables until the circuit clock is advanced again, at which point the information is recomputed.

8.1.2 Determining the Current Location in the Schedule

For the debugger to operate, it must know where the circuit is in the final schedule. Once the debugger determines the value of the state variables, as described above, Mapping L6:Sched-State is used to calculate the current schedule, which consists of the currently active block and the cycle of execution within that block.

8.1.3 Determining When a Predicate Equation Can Be Computed

Since the debugger has no way to know if the result of an operation will be committed until the predicate equation for the operation can be computed, it is necessary for the debugger to know when it is possible to compute the result of the equation. A predicate equation is computable once all of the predicates used in the equation have been computed. This means that the equation is computable on or after the cycle on which the latest predicate is computed. This makes it necessary to determine the cycle on which each of the predicates is computed. This is done by using Mapping V3:SSA-Op to find the operation which assigns the value of each of the predicates in the equation. Mapping L4:Op-FinalSched is then used to find when the operation is computed. Once this information has been determined for each predicate, the debugger selects the latest as the cycle on which the equation can be computed.

8.1.4 Determining If Predicate Equations Are Satisfied

Any time after a predicate equation is computable and before the variables become invalid (typically at the end of the block's execution) the debugger can determine if the equation is satisfied. This is done by querying the global repository of variable values

for the value of each of the predicates in the equation. The debugger then uses these values to compute the result of the predicate equation.

8.1.5 Getting the Final Schedule for an Original Operation

Another important mapping relates an original operation with the time in the final schedule when the computation is completed. This time corresponds to the schedule of the final operation which encapsulates the functionality of the original operation. It is computed by using Mapping L3:OrigOp-FinalOp to find the final operation which completes the computation. Mapping L4:Op-FinalSched is then used to find the schedule of that operation. In the case where the original operation maps to multiple final operations, then the minimum or maximum value is chosen, depending on what the result is needed for. The case where there is no final operation indicates that the original operation was optimized away.

8.1.6 Determining the Line Numbers Which Contribute to a Final Operation

Determining which line numbers have contributed to a final operation is needed in order to relate operation execution back to the source code. It is done by using Mapping L3:OrigOp-FinalOp to find all original instructions which have been used to create the final operation. If there are no original instructions, then the operation was not directly derived from a statement in the source code. For each of the original operations, Mapping L1:Line-Op is used to retrieve the source line from which it was derived. Duplicate entries in the list are then removed.

8.2 Implementing the Debugger Feature Set for Clock Step Mode

This section will discuss, in general terms, how the debugger uses the incremental mappings in the debug database and the larger mappings discussed in Section 8.1 to implement each of the features put forth in Chapter 1 for clock step mode.

8.2.1 Single Stepping

Because of the way Clock Step Mode is defined, single stepping is trivial and consists of simply advancing the hardware clock one cycle. However, if the rollback hardware described in Chapter 7 is inserted into the circuit, the debugger can also provide the user with the option to step backwards as well as forwards. The limitation on this is that the user cannot step back in time beyond the last checkpoint. Stepping backwards is accomplished by using the replay circuitry to return circuit state to the last checkpoint, then running the application forward until it is one cycle previous to where it was. This ability gives the user some added flexibility in controlling execution of the circuit.

8.2.2 Breakpointing

Breakpointing and single stepping are tied closely together. The way breakpoints are handled must be consistent with the definition of a single step. In the case of clock step mode a semantic breakpoint (see Section 2.1) is used. This means that a breakpoint is set immediately before any operations that are derived from the breakpointed line execute.

Breakpointing is an important part of any debugger. It allows execution to run full speed to a predetermined location in the source code. This allows the user to skip over large amounts of code in which they are not interested, without forcing them to single step over it.

During hardware execution, there are two possible ways to implement breakpointing: The first is for the debugger to single step the circuit, observing the state after each step and determining if a breakpoint has been reached. Implementing breakpointing in this manner is a straightforward, but undesirable approach because of the prohibitively long time it takes to readback the circuit state after every clock cycle. For this reason, only the second approach will be discussed here. This second approach is to add breakpoint circuitry to the synthesized hardware that is capable of stopping the execution of the circuit. This circuitry was discussed in Chapter 7. In the hardware debugger, breakpoints are only triggered in the active thread; all other threads are allowed to free-run, possibly

stopping after the active thread stops. This is done to avoid deadlock. This means that only breakpoints reached in the active thread will cause circuit execution to stop.

In debugging synthesized circuits, there are three distinct parts to breakpointing. The first part is determining where breakpoints can be set in the source code, the second is setting the breakpoint, and the third is running execution to a breakpoint.

Determining Where Breakpoints Can Be Set

It is useful to know before hand on which lines of source code breakpoints can be set. This information can be used to annotate the source level view. This type of annotation allows the user to quickly see which lines of code were optimized away by the debugger. The debugger determines which lines of source code contributed to the final application by first creating a list of all the final operations in the application and then using the method described in Section 8.1.6 to determine which source lines contributed to each of the operations. Duplicate entries are then removed from the list. The final list provides a tally of all line numbers which contributed to the final implementation of the application.

Setting Breakpoints

Because clock step mode uses semantic breakpoints, a breakpoint is set at the point where the earliest operation from the breakpointed line is about to execute. Earliest in this sense, means earliest in the final control-flow, not the original control-flow. To do this, the debugger refers to the debug database to determine which operation generated by that line of code executes earliest in the hyperblock. The breakpoint is then set on this operation. However, the breakpoint will only be triggered if the predicate equation for that operation is satisfied. Thus, the debugger must also know when the predicates will be computed. If the predicates are computed after the instruction is executed, then the user is warned that the breakpoint will trigger late and the actual breakpoint is triggered on the cycle when the final predicate is computed. Otherwise, the breakpoint is triggered on the cycle during which the instruction is computed.

To program the breakpoint unit, the debugger refers to the debug database to determine how the signals are wired to the breakpoint unit. It then creates the programming

data and programs the breakpoint unit based on this information. The process for doing this is outlined in detail below:

1. The specified line number is converted into a list of operations in the original control-flow that are derived from code on the line number. This is done by using Mapping L1:Line-Op.

2. For each operation in the list, the final schedule for the operation is determined using the method described in Section 8.1.5.

3. Given the final schedules for the operations in the list, choose the one that executes earliest as the operation on which execution is stopped.

4. Using Mapping L5:Op-Pred, the predicate equation for the breakpoint operation can be determined.

5. The debugger then determines when the predicate equation is computable.

6. The debugger determines on which cycle the circuit is to be stopped by picking the latest of the cycle when the predicate is computable and the cycle on which the operation executes. If the predicate equation becomes computable after the line of code executes, we refer to this as a late breakpoint. Execution will then stop on the cycle on which the predicate equation is computable. The difference between when the equation is ready and when the operation executes is known as the late cycles. This information can be given to the user.

7. The cycle and predicate information is then passed to Mapping L7:Op-BPU to get the programming data for the breakpoint unit. The breakpoint unit is then programmed with this data to set the breakpoint in the hardware. The exact manner in which the breakpoint unit is programmed is dependent on the actual hardware implementation.

Running Execution to a Breakpoint

Once the hardware breakpoint units are correctly programmed, running to a breakpoint is a relatively easy task. The first step is to enable the breakpoint units, if

necessary, then allow the system clock to free-run until the breakpoint unit stops circuit execution.

If a late breakpoint is set, then execution will stop some number of cycles after the actual breakpointed operation executes. However, it is also possible to be able to "stop" execution on the exact cycle that the operation executes. This is done by using the rollback circuitry described in Chapter 7. After the circuit has reached a late breakpoint, the rollback circuitry is used to return the state of the circuit to the last checkpoint. Since the debugger now knows that the predicate equation of the operation will be satisfied, an unconditional breakpoint is set for the cycle on which the operation executes and execution is allowed to run to this breakpoint. After execution is run to the unconditional breakpoint, the debugger reprograms the breakpoint unit with the original breakpoint, and debugging can proceed as normal.

8.2.3 Location of Current Execution Points

Because single stepping in clock step mode is based on cycling the hardware clock, it is necessary to provide the user feedback on which lines of code are currently executing in the hardware. To provide the greatest insight into how the execution of the circuit proceeds over time, the debugger displays the execution status of all operations in the active block. This is done by providing information about whether or not an operation has executed (referred to hereafter as execution state of the operation), and whether or not its predicate equation was met (predication state). There are three possible states for each of these. For operation execution, the possible states are: not yet executed, currently executing and previously executed. For predication state, the choices are: unknown, satisfied and not satisfied. The result is unknown if the predicate equation is not yet computable. To determine this information, the following steps are taken:

1. Determine the current schedule, consisting of the active block and cycle within that block by using the method discussed in Section 8.1.2.

2. Using Mapping L4:Op-FinalSched, create a list of all operations in the active block. Mappings L1:Line-Op and L3:OrigOp-FinalOp are used to determine which operations are derived directly from the source code; for those that are derived directly, determine from which source line they originate. Other operations are not considered, as they will have no bearing on what is displayed to the user in the context of the original source code.

3. For each operation in this list, determine the cycle on which it executes, as well as its predicate equation. This is done by using Mappings L4:Op-FinalSched and L5:Op-Pred, respectively.

4. The execution state of all the operations can now be assigned by comparing the schedule of the operation with the current cycle of execution.

5. The predication state of the operations is assigned by first determining if the predicate equation is computable. This is done with the method described above. If the equation is computable, then the debugger determines if the equation is satisfied or not. The execution and predication states of the operations are then annotated in the source level view on the appropriate lines.

The actual implementation in the debugger will determine how the information about the execution and predication states are displayed for the user. Section 10.1.1 shows how the Sea Cucumber Debugger displays this information for the user.

8.2.4 Watching Variable Values

Watching variable values in the presence of static single assignment is quite different from doing so in a typical software debugger. Because SSA expands a single variable into many different versions, there is no longer a single storage location for the value of the variable. For this reason, it is necessary to specify more information than just a variable name to set a watch; it is also necessary to know which version of the variable is of interest to the user. This can be determined in one of two ways. The most straightforward way is for the user to specify the desired version. This, however, requires

that the user be able to determine which version of the variable they want to watch. This can be accomplished by using the appropriate mappings to provide the user with a view of the final operations in the application.

The second approach is to allow the user to determine which variable is of interest by specifying the line number on which it is used and/or assigned. Because a variable can be both used and assigned on a single line, it is possible that there are multiple versions of the variable that could match. These variable versions are found by using the following steps:

1. Using the specified line number, Mapping L1:Line-Op can be used to find the original operations which are derived from that line of code.

2. Given these operations, Mapping V3:SSA-Op can be used to find all SSA variables used in this instruction. This list represents all versions of all variables which are used on the specified line.

3. The debugger then narrows this list of SSA variables by using Mapping V1:SourceVar-SSA to determine which of these are versions of the variable in question.

4. To find the final SSA variables which hold the values of interest, the debugger uses Mapping V2:InitSSA-FinalSSA.

After the correct variable version is found, the debugger can determine the value of the watched variable by querying the global value repository. If more than one variable matches the criteria, then the debugger can provide the user with all the values. In most cases it should be quite straightforward for the user to determine which value belongs with which instance of the variable on the line of source.

8.2.5 Setting Variable Values

The setting of variable values in the presence of SSA presents the same problems as watching of variables. In order to set a variable, the debugger must know which version(s) of the variable to set. This can be done in the same way as is done for the watching of variables: the user can specify the variable and version number or the variable and

line number on which it is used. If a line number is specified, then the debugger uses the method described above to find the correct variable version(s). If more than one version is found, the debugger simply sets all of the valid versions to the requested value. The actual value is set using one of the methods discussed in Chapter 4.

8.3 Conclusions

Clock step mode takes a hardware-centric approach to the control and observation of the synthesized circuit. By defining a single step as advancing the circuit clock a single cycle, clock step mode allows the user a more direct view of how the circuit operates, while still providing this information in the context of the original source code. This mode allows the user control and visibility similar to a software debugger, but different in that execution flow is parallel rather than sequential.

Another interesting feature of debugging hardware that takes advantage of parallelism and static single assignment, is that more state is kept in the running circuit. This is because the resources are not recycled as often. This means that once the mappings have been done, there is much more information that can be retrieved from the running application. This allows the debugger to provide the user with information about all instructions in the currently executing block. In contrast, a software debugger only provides information about what is currently executing. This additional state can also be used to step the execution backwards, as long as the appropriate support hardware is added to the synthesized circuit.

The next chapter will look at a software-centric approach to the debugger. This approach gives the user the illusion of sequential operation and attempts to make the execution of the parallel circuit appear as if the code were being run in software.

Chapter 9

Source Step Mode

Source step mode takes a software-centric approach to the definition of a single step. A single step is defined as allowing execution to advance one line in the source code, which leads the debugger to provide *expected* behavior (see Section 2.1). This approach simplifies the process of supplying intuitive feedback about the state of the circuit, but requires the more complex control of the circuit's execution by the debugger. This approach also does not give the user any insight into how the application was parallelized and optimized.

Source step mode requires the debugger to make the optimized, parallelized application appear to run sequentially in the order recorded in the source file. To accomplish this, the debugger uses a process I developed called Virtual Sequentialization. Virtual Sequentialization is the process of determining when an operation can be considered to have "executed", as well as determining which versions of SSA variables are valid for given ranges of operations. This chapter will first discuss how Virtual Sequentialization is accomplished. This will be followed by how this information can be used to enable the various features of the debugger.

9.1 Virtual Sequentialization

Virtual Sequentialization is the process of making parallel, reordered code appear to the user to execute sequentially (in the order specified in the source code) in the debugger. Under the assumption that only full blocks are merged, hyperblock formation preserves the original control-flow between hyperblocks and only reorders within a hyperblock. Therefore, Virtual Sequentialization takes place at the hyperblock level. Virtual

Sequentialization has three main pieces: determining operation execution order, determining valid variable versions, and determining the current operation given arbitrary circuit state. Each of these areas will be discussed below.

To provide an understanding of the terms used in the descriptions of Virtual Sequentialization, the main terms are defined here. These definitions will refer to the incremental mappings described in Chapter 6. For convenience, these mappings are again provided for reference in Table 9.1.

Basic Block Group A Basic Block Group is the collection of Basic Blocks which were merged together to create a hyperblock.

Root Basic Block Since hyperblocks are allowed to have only a single entry point, one Basic Block in all those merged together will be this entry point into the hyperblock. This block is called the Root Basic Block. A consequence of being a Root Basic Block is that there are no forward edges into the block from other Basic Blocks in the hyperblock.

Current, Past and Future Operations Current, Past and Future Operations refer to the operations found in the original control- and data-flow of the application, and are collectively known as Original Operations. The current operation refers to the operation that is currently being considered. A future operation is an operation which may occur after the current operation, and a past operation is an operation which exists along any valid path to the current operation, within the Basic Block Group. As an example, refer to Figure 9.1. If Op 13 is the current operation, then all operations in Blocks 1, 2 and 3 are past operations and Op 14 is a future operation.

Final Operations Final Operations refer to operations in the final control- and data-flow for the application.

Operation Schedule The Final Operation Schedule refers to the final schedule of operations. The final schedule consists of two parts. The first is the hyperblock in which the

Table 9.1: Incremental Mappings Found in the Hardware Line Number and Variable Tables

L1:Line-Op	Source Line $\longleftrightarrow$ Original Operation
L2:Op-InitSched	Original Operation $\longleftrightarrow$ Initial Schedule
L3:OrigOp-FinalOp	Original Operation $\longleftrightarrow$ Final Operation.
L4:Op-FinalSched	Final Operation $\longleftrightarrow$ Schedule.
L5:Op-Pred	Operation $\longrightarrow$ Predicate Equation.
L6:Sched-State	Schedule $\longleftrightarrow$ Circuit State.
L7:Op-BPU	Operation $\longrightarrow$ Breakpoint Unit Programming Data.
V1:SourceVar-SSA	Source Variable $\longleftrightarrow$ SSA Variable
V2:InitSSA-FinalSSA	Initial SSA Variable $\longleftrightarrow$ Final SSA
V3:SSA-Op	SSA Variable $\longleftrightarrow$ Operation/Instruction
V4:Var-Circuit	Variables $\longrightarrow$ Circuit Element
V5:SSA-Width	SSA Variable $\longrightarrow$ Hardware Width

operation can be found. The second is an integer that refers to a state number in the control state machine. This work assumes that the state machines are essentially sequencers which can potentially skip states, but which do not loop except to restart execution of the hyperblock. This means that for two operations in the same hyperblock, the one with the highest schedule will execute last. The schedule of Final Operations is provided by Mapping L4:Op-FinalSched. The final schedule of an Original Operation is computed as outlined in Section 8.1.5.

Operation Execution Operation Execution refers to the time in the final schedule during which an Original Operation can be considered to be "executing". Source step mode defines an operation as "executing" when the following conditions have been met.

1. All previous operations, along all possible control-flow paths, have been computed. This means that all previous operations in the current basic block have been computed, and that all previous basic blocks have computed results for all operations in the block.

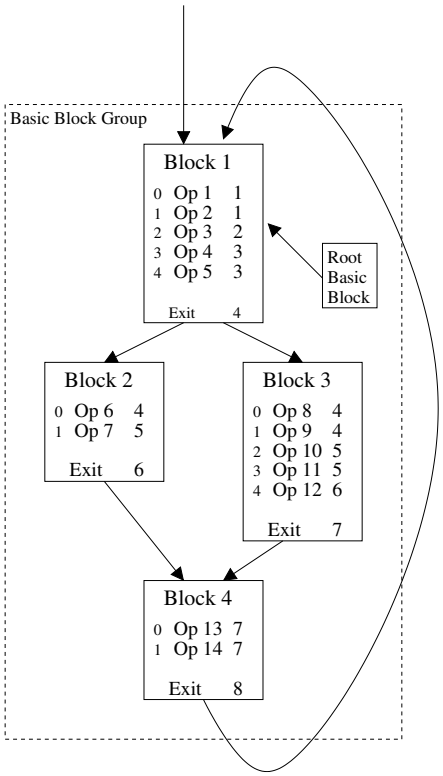


Figure 9.1: Sample Structure of Basic Blocks with Execution Times for Operations

2. All predicates used in the operation's predicate equation have been computed. This ensures that the debugger can guarantee that the results of the operation are actually used.

This ensures that all values of variables that are valid at the time the operation is executing have already been computed by the circuit. The synthesizer must ensure that each variable value remains valid until a new version of the variable is written, as discussed in Section 7.3. If these values are overwritten too soon, then buffering hardware must be added in order to keep the values available to the debugger.

This approach is taken to avoid the need to emulate the circuit to determine the values of variables. When using SSA in hardware, much of the past execution state is buffered in the registers which store the SSA variable values. This makes it more convenient to "over-execute" and use this buffered state to find variable values. This approach is opposite of that proposed in [18], which uses forward emulation to make an optimized program appear to run in the order specified in the source code.

9.1.1 Determining Operation Execution Order

Determining the order of operation execution within each hyperblock is done by using the conditions for execution outlined above. Doing this requires the debugger to correlate the original control-flow of the program with the control-flow of the final optimized program. This correlation provides the debugger with the following pieces of information, which are then used in simple equations to determine the execution time of each original instruction.

Original Operation Ordering (n) Mapping L2:Op-InitSched is used to determine the order of Original Operations in each Basic Block. The first operation to execute in each Basic Block is assigned $n = 0$. The value of n is then incremented by one for each subsequent instruction in the Basic Block as shown in Figure 9.1.

Schedule of the Original Operation (S) The final schedule of an Original Operation is computed as explained above.

Earliest Block Execution Time (p) A Basic Block cannot start executing until all of its source blocks have finished executing. This means that it is necessary to compute an "Exit" Time for each Basic Block. This time is computed by determining the cycle on which all operations in the block have executed. Figure 9.1 shows an example of these values. Exactly how this value is computed is discussed below.

In addition, no instructions in a block can execute until the predicate equation for the block is computable (determined by the method described in Section 8.1.3). In general, however, this case is already accounted for by the constraint that all previous blocks finish execution before the current block begins. This is because the predicates need to be computed before the correct branch out of the Basic Block is known.

The value p is computed by taking the latest of the Exit Times from all of the current Basic Block's source blocks. For example, in Figure 9.1, p for Block 4 would be the greater of 6 and 7, so $p = 7$.

Using this information, it is possible to compute the execution time for the n^{th} operation in a Basic Block (E_n), given the final schedule of each operation (S_n), by using the following equation:

$$E_n = max\{p, max\{S_0, ..., S_{n-1}\} + 1\}. \tag{9.1}$$

Although this equation will work to compute the execution time, it can be simplified by comparing the results for n and $n + 1$. If we put $n + 1$ in for n in this equation, we get:

$$E_{n+1} = max\{p, max\{S_0, ..., S_n\} + 1\}. \tag{9.2}$$

Pulling out S_n in the inner max operator we get

$$E_{n+1} = max\{S_n + 1, p, max\{S_0, ..., S_{n-1}\} + 1\}. \tag{9.3}$$

Wrapping the last two terms of the outer max operation with another max operation, we get

$$E_{n+1} = max\{S_n + 1, max\{p, max\{S_0, ..., S_{n-1}\} + 1\}\}. \tag{9.4}$$

We can now recognize that the inner max operator simply equals E_n. This gives the following equation:

$$E_{n+1} = max\{S_n + 1, E_n\}. \tag{9.5}$$

If we substitute $n - 1$ for n we get:

$$E_n = max\{S_{n-1} + 1, E_{n-1}\} \texttt{ for } n > 0. \tag{9.6}$$

Therefore, E_n is always greater than or equal to E_{n-1}. Using Equation 9.1 for $n = 0$ and Equation 9.6 for $n > 0$ we get the following equations for computing the execution time of operations in a basic block:

$$E_0 = max\{p\} = p \tag{9.7}$$

$$E_n = max\{S_{n-1} + 1, E_{n-1}\} \texttt{ for } x > 0. \tag{9.8}$$

Thus, the execution time for the first operation in a block is equal to the earliest execution time for the block. For all other operations, the execution time is the greater of the previous operation's final schedule plus one and the previous operation's execution time. These equations make it a very straightforward process to compute the execution times of all operations in a Basic Block.

After all operations in a Basic Block have been assigned execution times, the Exit Time can be assigned. The Exit Time for the block gives the earliest possible time that any operation following this Basic Block can execute. As such, the equation for determining the Exit schedule is the same as for determining the execution time for the next

instruction, if it exists. This gives the following equation for the Exit Schedule, which we will call X, for a block with m operations:

$$X = max\{S_{m-1} + 1, E_{m-1}\}. \tag{9.9}$$

To compute the execution times for all operations in a Basic Block Group, the debugger begins at the Root Basic Block. Since the Root Basic Block has no source blocks in the current hyperblock, p for this block is simply the latest schedule of the Predicate equation for the block. However, since the Root Basic Block is typically executed unconditionally, its predicate equation is normally $true$ and $p = 0$. Using Equations 9.7 and 9.8 the execution times of all operations in the Root Basic Block are computed. Equation 9.9 is then used to compute the exit time for this block.

The debugger then searches the graph in a breadth-first manner to ensure that each Basic Block has had the Exit Times of all of its source blocks computed. For these blocks p is computed in the way described above and the execution times of operations in each block and the Exit Time for the block are computed in the same manner as for the Root Basic Block. After the graph has been fully searched, all operations will have had their execution times computed.

9.1.2 Determining Valid Variable Versions

Once the execution time for each operation is determined, it is necessary to determine which version of each variable is valid while each operation is executing. Because storing this information for each operation would take a large amount of storage, it is best to simply recompute the information each time it is needed. This information can be computed in a straightforward manner using Mappings V1:SourceVar-SSA, V3:SSA-Op, and L2:Op-InitSched.

Finding the valid version of a variable is done by searching through the original operations to find the last operation which wrote to a version of the variable in question.

This is done by starting at the current operation and searching backward through past operations until an assignment to the variable in question is found. The result of that assignment is the version of the variable which is currently valid in the application.

This process can be accelerated by using a hashtable to store the latest variable assignments for each Basic Block. This eliminates the necessity of searching all the operations in each block, and is a good compromise between speed and storage.

Once the original variable version is found, Mapping V2:InitSSA-FinalSSA is used to find the final SSA variable which holds the correct variable value. Mapping V4:Var-Circuit will then give the location of that variable version in the circuit. This value can then be retrieved from the circuit state and reported in the debugger.

9.1.3 Determining the Current Operation For Arbitrary Circuit State

In general, source step mode controls the execution of the circuit such that the currently executing operation is known to the debugger. However, it is also necessary to be able to determine the currently executing operation given an arbitrary circuit state. This is needed in two different instances. The first instance is when the user desires to switch the debugger from clock step mode to source step mode. When this switch is made, the debugger determines the currently executing operation and then proceeds from that point. The second instance is when the debugger is providing information about threads which are not currently being directly controlled by the debugger. In general, it is only possible to control the execution of the active thread. All other threads simply run until the clock stops or a synchronization point is reached. When the debugger reports information about the non-active threads, it must determine the current operation based on the state of the thread.

Since it is possible for many operations to have execution times occurring at the same time in the schedule, it is necessary to determine which of these operations will be considered to be executing. While it is possible to say any of these is the current execution point, the algorithm presented here chooses the latest operation possible. This is done by finding the latest original operation which fulfills the following criteria:

1. The operation is contained in the currently active hyperblock.

2. The execution time of the operation is less than or equal to the current execution schedule of the active hyperblock.

3. The operation's predicate equation is computable and satisfied by the current state of the predicates in the circuit.

9.2 Implementing the Debugger Feature Set for Source Step Mode

This section describes the underlying mechanics of how the feature set for the hardware debugger can be implemented. These procedures are meant for reference only, actual implementations can use this information as a starting point and make changes as necessary. To see the specifics of how these features are implemented in the Sea Cucumber Debugger, see Chapter 10.

9.2.1 Single Stepping

Single stepping in source step mode is more complicated than for clock step mode. Source step mode uses information about the original control-flow of the program to control the execution of the final circuit. Before the debugger can provide single stepping, it must first determine the execution time for each operation as discussed in Section 9.1.1. When a single step is requested, the debugger must determine which operation is the next operation to which the debugger should step. This is done by searching forward from the current operation until the next operation not derived from the same source line is found. If this operation is not found before the end of the basic block is reached, then the debugger advances the clock to the end time for the block. This is done to ensure that all predicates in the block are computed, which makes it possible to determine which basic block is next in the control-flow. The debugger then searches in that block in order to find the next operation.

Once the next operation to execute is found, the debugger changes its internal execution pointer to the new operation and advances the clock to the execution time of that operation. If the next operation time is equal to the location in the schedule, then the clock

does not need to be advanced. If the hardware clock does need to be advanced, there are two approaches which can be taken. In the first approach, the debugger must have some information about how the state machines in the hardware operate. With this knowledge, the debugger can determine how many cycles the hardware clock must be advanced to reach the new execution time. The clock is then advanced that many cycles. A better approach is to set an unconditional breakpoint for the cycle on which the operation executes and allow the circuit to run to that point. This approach not only removes the requirement that the debugger understand the state machine execution, but also works when it is difficult to determine how many cycles to advance the clock.

Just as in clock step mode, the hardware debugger can allow the user to step backward in source step mode. This is done by searching backward for the operation on the previous line which executes earliest. The internal pointer in the debugger is then set to this operation. Unlike in clock step mode, the circuit does not actually need to be moved backward; as long as the buffered state for the previous line is still valid, the debugger simply uses the internal pointer to determine which variable values are valid.

9.2.2 Breakpointing

The breakpoints used in source step mode are syntactic breakpoints (see Section 2.1). This means that a breakpoint is triggered at the point where all operations previous to the one on which the breakpoint was set have executed. This is consistent with the definition used for single stepping in source step mode.

Just as in clock stop mode, implementing breakpointing in source step mode has three main challenges: determining where breakpoints can be set in the source code; setting the breakpoint; and running execution to a breakpoint. Determining where breakpoints can be set is done in the same way as it is done is clock step mode. The others, however, have to be considered differently when in source step mode.

Setting Breakpoints

Before breakpointing can be implemented, the debugger must determine the execution time for each original operation as discussed in Section 9.1.1. When a breakpoint

is set in source step mode, the debugger uses Mapping L1:Line-Op to determine the operations derived directly for the source line on which the user desires to set a breakpoint. Just as with software debuggers, the breakpoint is set on the earliest operation derived for the breakpointed line. Mapping L2:Op-InitSched is used to determine which of these operations is the earliest. Just as with breakpointing in clock step mode, the chosen operation must be mapped to a cycle and predicate equation. The cycle used is the execution time of the operation and the predicate equation is the predicate equation of the basic block in which the operation is found. This information is then used to get the programming data for the breakpoint unit by using Mapping L7:Op-BPU. Once this data is created, the breakpoint unit can be programmed.

Running Execution to a Breakpoint

Just as in clock step mode, the debugger runs to a breakpoint by enabling the breakpoint units and allowing the clock to run until a breakpoint is triggered. However, in source step mode, there is no such thing as a late breakpoint. This is because an instruction cannot be considered to be executing until its predicate equation is computable.

After the breakpoint is triggered, the debugger determines which breakpoint was triggered and sets the current operation in the active thread to be the operation on which the breakpoint was set. For other threads, the current execution point is determined as discussed below.

9.2.3 Location of Current Execution Points

For the active thread, the current execution point is controlled by the debugger. For all inactive threads, the debugger must determine where the current execution points are. This is done by first determining the current schedule location for each thread. This is done using the method in Section 8.1.2. This information is used in conjunction with the method found in Section 9.1.3 for determining the current operation from arbitrary circuit state. This operation is then used as the current operation for determining the currently executing line, and for watching and setting variable values.

9.2.4 Watching Variable Values

Watching variable values in clock step mode is, in one respect, more direct than in clock step mode. For instance, watching a variable does not require the user to specify a particular variable version. Rather, the debugger can determine which version is the correct one based on the currently executing operation. This is done using the method described in Section 9.1.2. Once the correct version of the variable is determined, the debugger finds the value of the variable by querying the global repository of variable values. This information is then presented to the user.

9.2.5 Setting Variable Values

Setting variables in the presence of reordered execution creates some interesting issues. It also allows for a wide range of possible behaviors. At one end of the behavior spectrum, variables can simply be set without regard to operation scheduling; at the other end, the debugger can ensure that the setting of variables behaves exactly like in a software debugger. The first option is undesirable, as it creates unpredictable results. While the second option may appear to be desirable, it can be very difficult and costly to implement. This work takes a compromise approach that provides advantages both in functionality and implementation. The main issues involved with the first two approaches will be discussed and the compromise approach will be discussed in detail thereafter.

Setting Variables without Regard to Operation Scheduling

If the debugger does not take operation scheduling into account, then the setting of variables is very straightforward to implement. The debugger simply needs to determine which version of the desired variable is currently the correct one and set that version to the given value. However, this approach can have unpredictable results for the user. It is quite likely that operations from the current or future lines have already executed and are not affected by the value change. Such results are counter-intuitive in an environment which is trying to emulate sequential execution.

```
20 a = b + c;
21 d = a + e;
22 b = c + f;
```

(a) Source Code

```
add a_2 b_1 c_2   (S = 2)
add d_3 a_2 e_1   (S = 3)
add b_2 c_2 f_2   (S = 2)
```

(b) Final Operations with Schedule

Figure 9.2: Sample Code for Variable Setting. This code demonstrates one of the reasons why it is difficult to set variables directly in hardware when full regard to operation scheduling is taken.

Setting Variables with Full Regard to Operation Scheduling

Setting variables with full regard to operation scheduling means that changing a variable value will affect all operations on the current and future lines and no operations from previous lines. This can be accomplished one of two ways. This first is to limit the code locations where variables can be set. These locations would correspond to the start of execution for each hyperblock. At this point, the original and final-control flow coincide. This approach, however, greatly reduces the number of places a variable value can be set.

The second is to allow the user to set variable values at arbitrary code locations. In this case, it is the responsibility of the debugger to ensure that the correct operations are affected or not affected, as appropriate. Implementing this behavior can be very difficult, and is probably best done by emulating the effects and propagating them to the hardware. As an example of why doing this in hardware can be difficult, consider the source code with its corresponding final operations and schedules listed in Figure 9.2.

To demonstrate the difficulty, suppose that the current line of execution is line 21 and the user desires to set the value of variable c. The problem lies in the fact that both the add on line 20 (which should not be affected by the value change) and the add on line 22 (which should be affected by the value change) use the SSA variable c_2 as input. Further, these two operations execute on the same clock cycle in the final circuit, making it impossible, without the addition of extra circuitry, to affect one operation and not the other.

This is just one example of the difficulties which arise with implementing the setting of variables in this way.

Setting Variables Using a Compromise Approach

A compromise approach balances flexibility in use with ease of implementation. Another advantage is that it can improve upon the approach used in software for the setting of variables. Modifying variable values with a source level debugger is problematic because it is difficult to avoid presenting the user an inaccurate or misleading perception of the effects caused by the modified values. This occurs because variable values can be changed by the debugger at code locations other than the locations in the source where the variables are assigned a value. This can lead to incoherent results.

One way to avoid this incoherency is to only allow the user to set variables immediately after the program has set the value and before that value has been used. In this case, all statements will see the same value for the variable. However, in practice this is very limiting. The Compromise Approach seeks to remove the limitations, while keeping the results coherent. The debugger does this by using the replay circuitry discussed in Chapter 7. Using replay, modifying values through the debugger is achieved as follows. First, circuit execution is run back to the last cycle where the variable received its most recent value. Second, the variable is set to the new value as specified by the user of the debugger. Finally, the circuit is executed forward from this point, up to the point where the process started. This approach ensures that all operations will see the exact same value for the variable, keeping the execution coherent.

In practice, it is not always possible to run execution back to the last assignment of a variable. This happens when the last assignment to the variable to be set was earlier than the last checkpoint. In this case, the execution is run back to the last checkpoint and the value of the variable is changed at this point, then executed forward as explained above. This will guarantee coherency for the current block (assuming checkpointing is done at a block boundary).

If the user desires to set a variable, the debugger uses mappings V1:SourceVar-SSA, V3:SSA-Op, and L2:Op-InitSched as described in Section 9.1.2 to find the current

valid version of the variable in the context of the original control- and data-flow. Mapping V2:InitSSA-FinalSSA is used to determine the final SSA variable which holds the value of the variable. If this SSA variable was computed since the last checkpoint, then execution is run back to the last checkpoint and then forward to the cycle just after the SSA variable was last assigned. If the SSA variable was computed prior to the last checkpoint, then execution is run back to the last checkpoint. At this point, Mapping V4:Var-Circuit is used to determine the location of the variable in hardware and the value is changed to that requested by the user. Execution is then run forward to its previous point.

Another issue can arise when the user sets a variable which is used to compute a predicate. Changing such a variable may change a predicate state and make it impossible to return execution to the original point. If this happens, there are two main options. The first is to return the circuit to the same state as it was previously, but now, execution will no longer be where it started. The second is to leave the execution at the location just after the variable was set. In either case, the debugger should notify the user that a predicate value was changed by the user's request to set a variable, making it impossible to return execution to the original location.

9.3 Conclusions

Source step mode allows a user of the hardware debugger much the same control style and visibility as a software debugger. The user can single step, set breakpoints and watch and set variables in the same manner as in a software debugger. The primary exception is that the hardware debugger cannot execute instructions or watch or set variables which have been optimized away. In order to provide the same functionality and execution style as a software debugger, the hardware debugger uses Virtual Sequentialization to make the reordered, parallel circuit appear as if it were running sequentially in the order presented in the source code. While this mode is more difficult to implement, this approach makes it easier to debug the functionality of the circuit, but provides no insight into how the application was optimized.

Because the synthesized circuit is able to preserve and observe more state than its software counterpart, the hardware debugger is able to improve the functionality of

setting variables. It does this by providing more consistent behavior for the setting of variables in arbitrary locations in the code. This is accomplished by retroactively setting the variable state such that the value change is reflected in as many operations as necessary.

The two modes of the debugger are complementary and can be used by a programmer to debug the functionality of the program, as well as gain insight into how the application was mapped to the final hardware. The ability to change between these two modes provides the user with more options and flexibility during debug.

Chapter 10

Sea Cucumber Debugger

This chapter will detail the Sea Cucumber Debugger's user interface and internal operation. The Sea Cucumber Debugger uses the methods described in the previous chapters to provide the user with a source-level view of a circuit synthesized from Java bytecode by the Sea Cucumber synthesizing compiler. The SC Debugger provides both clock step and source step modes for use. While the primary purpose of the debugger is to study the feasibility of providing information about a synthesized circuit running in hardware in the context of the original source code, there is no compelling reason to hide the circuit-level details from the user. For this reason, the debugger uses JHDL to provide the user with the typical views used to debug hardware at the circuit level. A block diagram of how SC, the SC Debugger and JHDL interact is shown in Figure 10.1.

This chapter will first discuss the user interface for the debugger, and how the data for these views are generated. Then the platform interface API used to interface the debugger to the JHDL simulator and to hardware will be discussed. In general, the SC Debugger adheres closely to the methods described in Chapters 8 and 9. Exceptions and clarifications are noted during the appropriate discussions.

10.1 User Interface for the Sea Cucumber Debugger

Figure 10.2 shows a screen shot of the Sea Cucumber Debugger. The debugger uses the Application Framework provided with JHDL [60] to provide a unified format for both the source- and circuit-level views of the application. During work on the SC Debugger, I found it necessary to enhance the features of the viewers available in JHDL. These enhancements include versions of all viewers that can be used inside a Java desktop

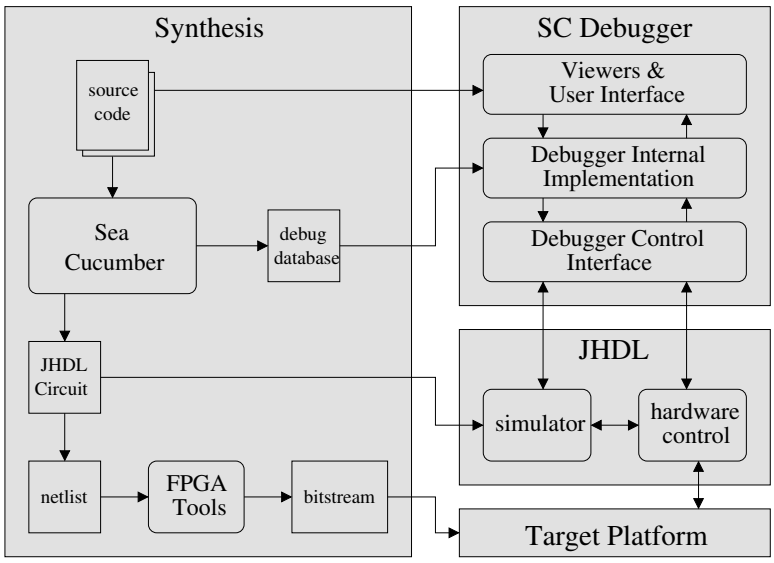


Figure 10.1: Block Diagram of the Interactions Between SC, the SC Debugger and JHDL

window, as well as a resource manager for JHDL. The first enhancement was to allow the debugger to use a unified desktop for the source- and circuit-level views. The second was to allow the user to chose the colors used in the main application as well as all the viewers. This was important because much of the feedback given to the user is provided through different color annotations. The ability to change these allows the user more flexibility to create intuitive color schemes. The desktop enabled windows, along with the color selection window can be seen in Figure 10.2.

As can be seen in the figure, the debugger is divided into 3 main parts: the toolbar on the top, the control window on the left and a Java desktop on the right. The toolbar provides control over single stepping, stepping backward, and running to a breakpoint, as well as the ability to change between clock step and source step modes. It also provides a textual interface for controlling the debugger. The textual interface is an instance of the command line interpreter (CLI) included with JHDL.

The control window provides the user with the hierarchy of the application. The top view provides a thread-oriented view of the program. The bottom view provides access

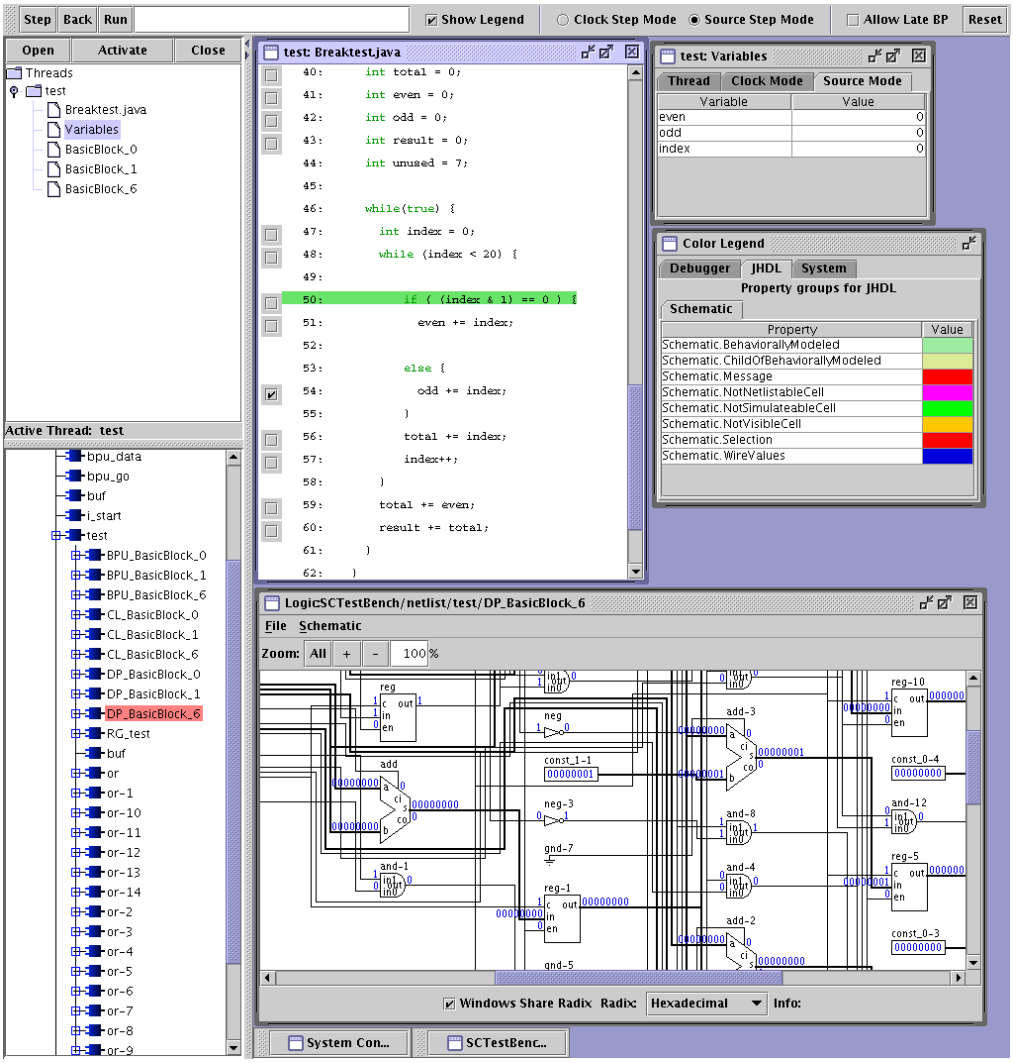


Figure 10.2: The Sea Cucumber Debugger

to a tree view of the circuit hierarchy. These panels can be used to open other views of the application. All the other views appear in separate windows in the desktop on the right.

Following the guidelines of the Application Framework, the graphical interfaces of the debugger produce text commands which are passed to CLI. This makes it possible to control the circuit both in the graphical views, as well as textually, and allows the user to script commands.

The debugger provides four main views of the circuit state: source view, variable view, assembly-level view and circuit-level view. Each of these views will be discussed below.

10.1.1 Source View

Each thread in the program is ultimately derived from Java source code[1]. The `Source Viewer` provides the user with a view of this code. The `Source Viewer` provides the user both with feedback from the running circuit and with the ability to control certain aspects of circuit execution. The main purpose of this viewer is to provide the user with feedback on which operations are currently executing in the hardware. In clock step mode, the SC Debugger provides feedback in a slightly different manner than that described in Section 8.2.3. Rather than separately providing information about execution state and predication state, the `Source Viewer` uses a single code highlighting scheme to provide feedback about both. This is accomplished by only providing information about the predicate state after an operation has executed. This is done by separating each operation in the active hyperblock into one of five groups, shown below:

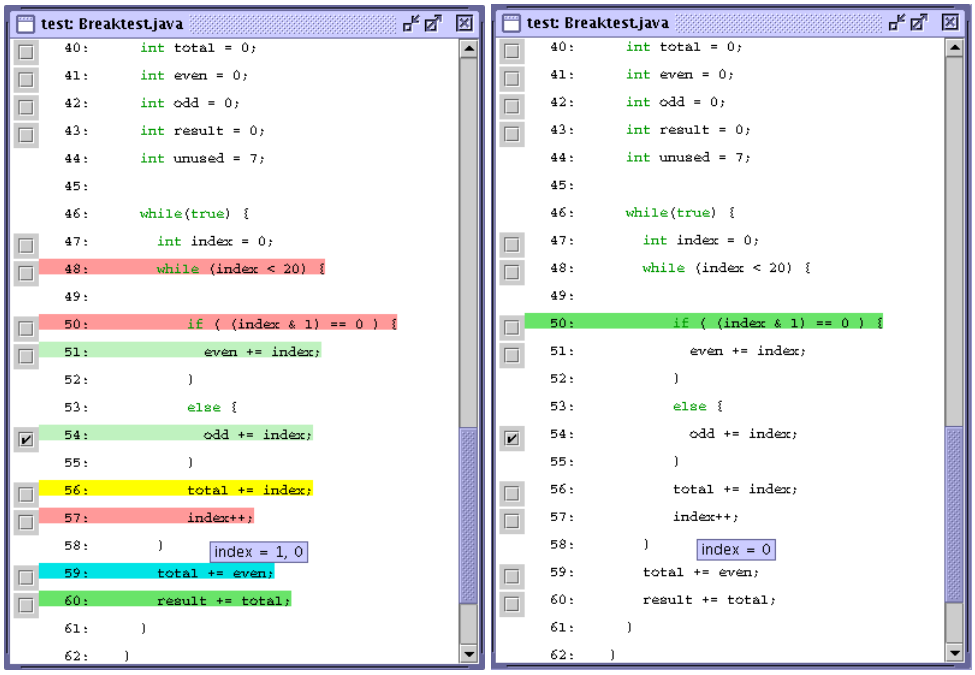
1. Operation not yet executed
2. Operation currently executing
3. Operation executed, but predicate state unknown
4. Operation executed, predicate equation satisfied
5. Operation executed, predicate equation not satisfied

In source step mode the `Source Viewer` simply highlights the source line which is currently considered to be executing. An example of the highlighting for both modes can be seen in Figure 10.3.

The `Source Viewer` also provides the user with the values of variables in the thread. This is done by placing the mouse pointer over the variable of interest. After a

[1]Technically, the synthesized circuit is derived from Java bytecode. However, the bytecode was generated from the Java source code which is displayed in the debugger.

short delay, the value of the variable will appear, as seen in Figure 10.3. The behavior is slightly different for the two different debugger modes. In clock step mode, the debugger will display the value of all the versions of the selected variable that are used on the current line. This is done because the debugger can only differentiate code on line boundaries, as this is all the information the original bytecode supplied to the synthesizer. In source step mode, the debugger hides the details of the different versions, and based on the currently executing operation, can chose the correct version of the variable and display only that single value (see Section 9.1.2).



(a) Clock Step Mode (b) Source Step Mode

Figure 10.3: The Sea Cucumber Debugger Source Viewer

The `Source Viewer` also allows the user to set breakpoints and variable watches. Breakpoints are set by clicking on the checkbox to the left of the line on which the user desires to set a breakpoint. In clock step mode, the debugger will warn the user of any late cycles for a requested breakpoint. Optionally, the user can choose to disallow late breakpoints. In this case, the debugger uses the rollback circuitry as described in Section 8.2.2 to stop the circuit on the correct cycle. A check in the checkbox indicates that a breakpoint is currently set on the associated line. This breakpoint can be removed simply by clicking the button a second time. Breakpoints can also be added and removed using the command line interpreter.

A variable watch popup menu can be created by right-clicking the variable of interest, as shown in Figure 10.4. In clock step mode, this will create a menu which allows the user to select a watch on any of the variable versions available on that line. These versions are determined by the method described in Section 8.2.4. In source step mode, the menu will provide only one option, as determined by the method in Section 9.1.2. When a watch is selected, the variable is added to the `Variable Viewer`, which is discussed next.

10.1.2 Variable View

The `Variable Viewer` provides a single location for viewing multiple variable values and is an alternative to mousing over the variable in the `Source Viewer`. The `Variable Viewer` is pictured in Figure 10.5 and consists of three parts. The first part provides the current value of each of the variables found in the thread registers. The user is not allowed to add or delete variables from this view. The second view is a view of variables watched in clock step mode and the third provides the values of variables watched in source step mode. These last two views are separated to make it easier to switch the debugger between clock step and source step modes.

The `Variable Viewer` also provides the user with a method of setting the value of a variable. For variable versions which are settable, the value field of the variable table is editable. If the user edits the value of the field, then the debugger will set the value of the corresponding variable as directed. The only other way to set the value of the variable

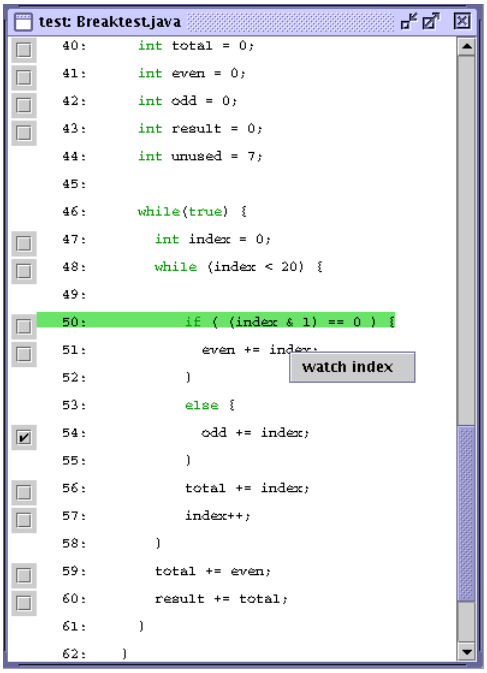


Figure 10.4: The Sea Cucumber Debugger Source Viewer with Variable Watch Popup

is through a textual command. For clock step mode, both of these methods require the user to specify the exact version of the variable to be set. This is facilitated by the use of the assembly level view, discussed next.

10.1.3 Assembly-Level View

The assembly-level view provides the user with information about how the original code was mapped to hardware. The SC Debugger allows the user to look at the final operations in each hyperblock, as shown in Figure 10.6. This information is particularly useful when trying to determine which version of a variable to watch or set in clock step mode. It also allows the user to see the results of the optimizations performed by Sea Cucumber.

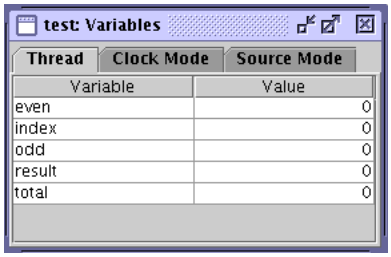


Figure 10.5: The `Variable Viewer` in the Sea Cucumber Debugger

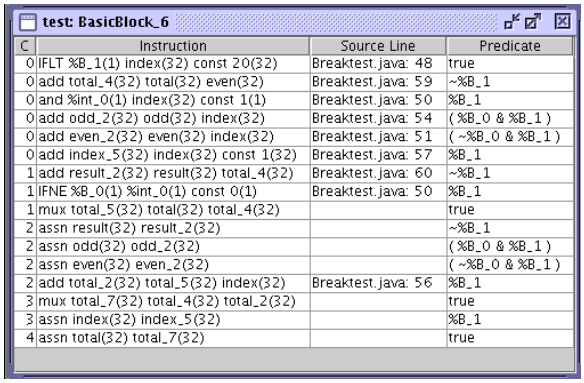


Figure 10.6: Assembly-level Views in the Sea Cucumber Debugger

10.1.4 Circuit-Level View

The SC Debugger allows the user access to all of the default circuit viewers which are available with JHDL [61]. This includes, but is not limited to, the following: `Tree Viewer`, `Cell Viewer`, `Waves Viewer`, and `Schematic Viewer`. The `Tree Viewer` is the panel seen on the bottom left in Figure 10.7, and provides access to the other viewers. The other viewers can also be accessed through the command line interpreter. Figure 10.7 also shows the `Waves Viewer` at the bottom of the desktop, the `Schematic Viewer` in the middle and the `Cell Viewer` in the top right corner.

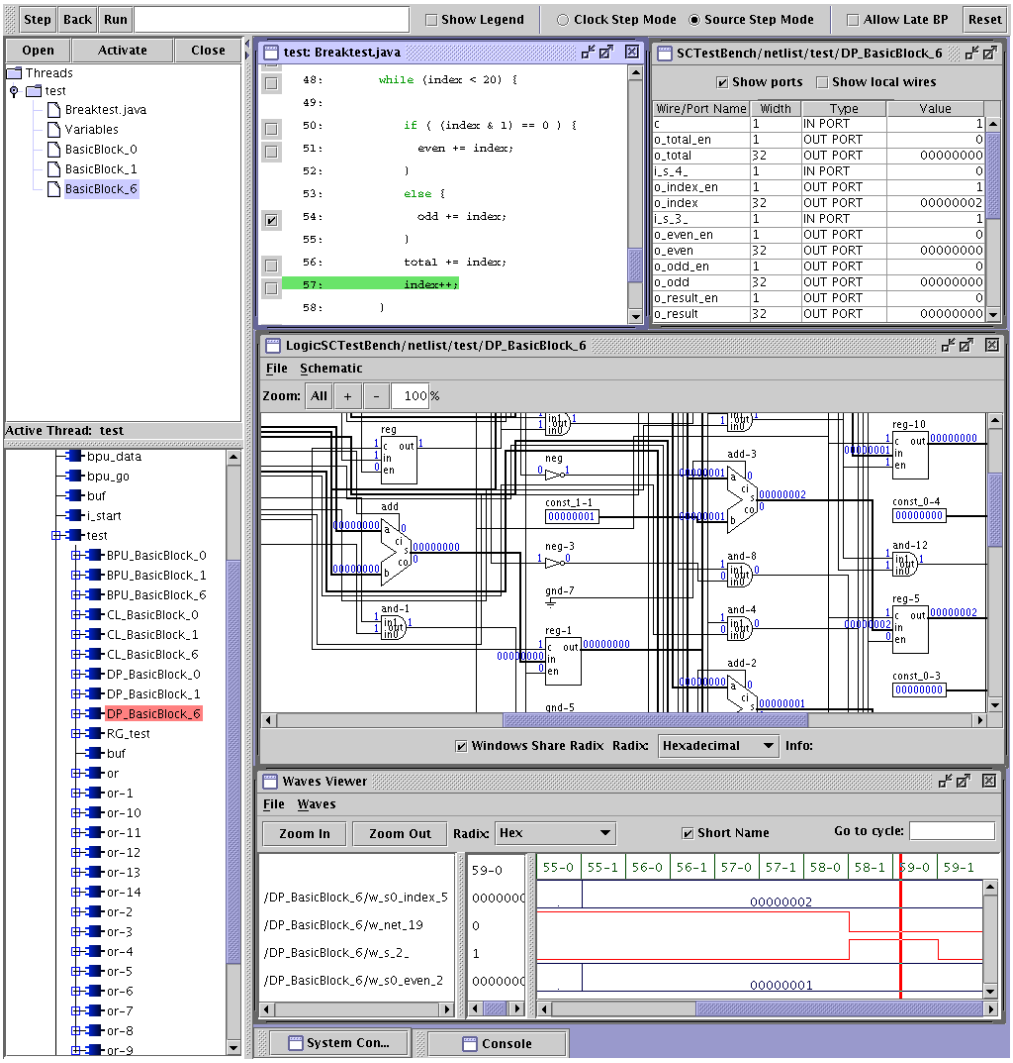


Figure 10.7: Circuit-level Views in the Sea Cucumber Debugger

10.2 Platform Interface API

As was discussed in Section 2.2.2, the debugger uses JHDL as both a simulation and hardware control layer. This means that the debugger uses the same API to interact with both simulation and hardware execution. In conjunction with the use of JHDL as the interface layer, the debugger also defines a higher level interface which allows it to work transparently with different target platforms. This is done by using a Java interface

```
void setDebugSignalValue(int signal, int value);
ST_Program getDebugDatabase();
Cell getTopLevelCell();
Wire getBreakpointHit();
void resetCircuit();
void freeRunClock();
void stopClock();
boolean hasHitBreakpoint();
void setReadBack(boolean state);
void hardwareCycle(int cycles);
```

Figure 10.8: API Calls for the `CircuitDebuggerInterface`

[56] called the `CircuitDebuggerInterface`. The interface performs two main functions. The first is to cause the SC intermediate netlist format to create a JHDL circuit. The `CircuitDebuggerInterface` can also create any platform-level circuitry that must be added to interface the synthesized circuit to the target platform.

The second responsibility of the interface is to provide method calls which allow the debugger to get information about where to find the synthesized circuit in the JHDL hierarchy, as well to provide methods to control the debug circuitry during runtime. The API is quite simple and consists of the methods listed in Figure 10.8.

The debugger was designed such that it is a simple matter to switch which `CircuitDebuggerInterface` is used during a given run of the debugger. The user can specify which interface to use when the debugger is started, and there is a default interface which is used if none is specified. The default `CircuitDebuggerInterface` and the specific one used to interface to the Slaac-1V board are discussed below.

10.2.1 Default CircuitDebuggerInterface

The default `CircuitDebuggerInterface` is `SCTestBench`. This class implements a JHDL testbench [61]. The circuit built by the synthesizer is the only cell built within the testbench. The debug signals are controlled directly by the testbench so that the debug circuitry is simulated along with the rest of the circuit. Operating with this

interface works only in simulation. If hardware mode is desired, an appropriate `CircuitDebuggerInterface` must be created for the target hardware platform.

10.2.2 Slaac-1V

The `CircuitDebuggerInterface` targeting the Slaac-1V board is named `Slaac1vTestBench`. Contrary to what the name might imply, the `Slaac1vTestBench` does not implement a JHDL testbench, as the default interface does. Rather, the class simply instances the JHDL Slaac-1V board model and builds the synthesized circuit in X1. The `Slaac1vTestBench` also inserts additional circuitry in X0 to pass the debug signals, which originate from the registers provided in the X0 register interface, through to the design in X1.

The calls made through the `CircuitDebuggerInterface` are translated to the proper register read and write calls. This interface can be used both for simulation and for hardware mode. Hardware mode is enabled simply by turning it on in the debugger and the JHDL core. Readback is enabled for X1 so that the state can be retrieved from the synthesized circuit.

10.3 Limitations of the Debugger

Because the SC Debugger was created to determine the feasibility of implementing a hardware source level debugger, there are some limitations in its feature set. The main limitation is that the setting of variables has not been implemented in hardware mode. The setting of variables has been verified using the Slaac-1V JHDL simulation environment. There have been implementations of setting variable values in hardware mode (these were discussed in Chapter 4). To enable this would require implementing one of those methods, which does not present any new research value in the context of this work; verifying the operation of the software and inserted debug circuitry in simulation is sufficient to show that such an implementation is feasible.

10.4 Conclusions

The creation of the Sea Cucumber Debugger has shown that it is possible to provide typical software debugging features in a hardware source level debugger. This includes the ability to single step, breakpoint, and view and modify the values of variables. The Sea Cucumber debugger supports both clock step and source step modes. This provides the user with expected or truthful behavior during execution and allows the user to switch between the two on the fly, so that the advantages of both modes are available. The debugger also provides intuitive feedback about and control of the operating circuit in the context of the original source code. This increases productivity as the user does not have to deal directly with the synthesized circuit.

The use of the debugger greatly enhances the users ability to debug circuits generated by Sea Cucumber, and passes on the same speed advantages to debugging that the synthesizer passes on to mapping the application to hardware. This increase in speed can help to decrease the time spent in debugging and verifying a circuit.

Chapter 11

Conclusions

Due to the rapidly increasing size of modern FPGA devices, design, debug and verification of FPGA circuits is becoming increasingly complex. This has lead to the search for better tools and techniques to accomplish these tasks. This dissertation has focused on a tool to aid in the debug of FPGA circuits synthesized from high level descriptions. The main contribution of this project was to demonstrate the feasibility of providing controllability and observability of an executing FPGA circuit in the context of the original high level source code. This goal was accomplished through the creation of the Sea Cucumber Debugger.

The SC Debugger allows a user to control and observe the hardware execution of a synthesized circuit in the context of the original Java source code. Providing this type of controllability and observability allows the user to debug a synthesized circuit at the same level of abstraction in which the circuit description was written. However, enabling this level of controllability and observability requires support from both synthesis tools and the target hardware platforms. These requirements are discussed in the following sections.

11.1 Hardware Debugging

As compared to software debugging, most of the techniques used to debug synthesized circuits seem ad-hoc at best. This discrepancy arises because of the lack of standard hardware debuggers. In contrast, software debugging is a well known, if not well understood, problem, and there are many standard software debugging platforms available today. This is possible because of the existence of standard computer architectures. In contrast, creating a standard hardware debugger for FPGA platforms is not currently

possible because of the lack of a standard feature set for FPGA platforms. It is also complicated by the fact that standard FPGA design tools do not provide enough information about how a design is mapped to hardware. The following sections will review the platform requirements and the mappings that must be provided by design tools to enable hardware debugging of FPGA circuits.

11.1.1 Hardware Requirements

Though the hardware features required to enable hardware debugging are straightforward, there are many FPGA platforms which do not provide all the necessary features. This section will review the device and platform features required to provide the controllability and observability necessary to enable hardware debugging. For purposes of this work, the FPGA platform includes the FPGA devices, supporting circuitry (such as memories, etc.), board firmware and software control libraries. The main features required for hardware debug are clock control, device state visibility, device state setting, and a scheme for communicating with the embedded debug hardware. These features can be supplied by any part, or combination of parts, of the platform, however, if these capabilities are not present in a particular platform, it is possible, in some cases, to add additional hardware to the synthesized circuit to support the feature. However, it is always desirable for these features to be provided as part of the FPGA platform.

Clock Control

Execution control is a fundamental part of debugging an FPGA circuit. To provide execution control, an FPGA platform must provide a mechanism to control the circuit clock. This support includes starting, stopping and stepping the circuit clock. For platforms which have multiple FPGAs on the board, it is important that the stopping and starting of the clock is synchronized across all FPGAs. This synchronization is difficult to accomplish through the addition of clock stopping circuitry to the FPGAs.

Device State Visibility

For the debugger to provide feedback at the source level, it must be able to access the full state of an executing circuit. To provide full visibility into the state of the circuit, there must be a mechanism in place to extract the circuit state from an FPGA without upsetting that state. This support is best built directly into the FPGA device, as instrumenting a design with this capability incurs too large an overhead in resources to be effective [40]. The Xilinx XC4000, Virtex and Virtex 2 families of FPGAs provide a feature known as readback which allows the state of the device to be read serially from the device. Even though the devices themselves support readback, many FPGA platforms based on these devices do not provide easy access to this feature.

Another aspect of providing full state visibility is found in mapping logical circuit elements to physical locations in the readback bitstream. Though this work was able to build on the solution found in [25], it would be preferable for the FPGA tool vendors to provide this information as part of the report files generated when creating a programming bitstream. The main advantage of having the logical to physical mapping generated by the vendor's FPGA tools is that these mappings would always be available, even when the tools or devices are modified. Such modifications currently necessitate the third party tool (in this case JHDL) to be modified to work with the new tool or device each time the tools are modified or when a new FPGA architecture is introduced.

Setting Device State

The ability to set the state of an executing FPGA allows the debugger to set the values of variables in a running circuit. If the user only desires to observe the state of variables, but not set them, then this capability is not required. However, this ability can also be used to control the debug circuitry that is added to the circuit. Device state setting can be implemented on some modern devices through bitstream manipulation. This is done by manipulating the power-on state of circuit element in an FPGA [40].

The ability to set the value of specific resources also requires a mapping from the logical circuit element to the physical location on the FPGA. As with device state visibility, this mapping is best provided by the FPGA tool vendor. In addition, it would

be advantageous for the FPGA vendor to supply a tool which can modify the bitstream to insert the desired state into the circuit. JBits is the only known example of such a vendor-supplied tool. However, JBits is a non-standard tool, has an uncertain future, and requires an NDA.

Communication with Debug Hardware

The same facilities for setting state on the FPGA can be used to communicate with the embedded debugging circuitry. However, it is overkill and a separate facility such as JTag or some other standard interface is usually more appropriate. The only caveat is that the chosen method not interfere with the state of the executing circuit.

11.1.2 Compiler/Synthesizer Requirements

As stated previously, the debugger cannot function without information from the synthesizing compiler. This information allows the debugger to correlate circuit state with elements in the source code. In general, tool vendors have been reluctant to provide these mappings for fear of unwittingly revealing important intellectual property or trade secrets. However, until tool vendors can agree to provide a standard set of mappings, it is impossible to create a set of standard debuggers for source level debugging of circuits synthesized from high level descriptions.

Another, less desirable option, is for tool vendors to provide a debugger with their compiler and/or synthesizer. This would allow them to provide the mappings to the debugger in a proprietary fashion which does not endanger any trade secrets. However, this means that the user would be locked into using a specific debugger when using such proprietary tools.

Recognizing the vast differences between different compilers and synthesizers, the goal of having standard mappings that would work for all the various tools may never be realized. The actual mappings may need to be modified based on the type of optimizations made during the compilation and synthesis processes. However, as long as the mappings are openly available, standard debuggers can be upgraded to include their use.

The current work, which deals with debugging circuits synthesized using predication, static single assignment, block merging and instruction scheduling, provides the mappings for the debugger as twelve incremental mappings. These mappings are combined by the debugger to make complete mappings from source code to circuit elements. These mappings are representative of those that would need to be supplied by any synthesizing compiler, in order to enable source level debugging of synthesized circuits..

The twelve incremental mappings are broken into two groups. The first contains mappings which provide information about the control-flow of the application; the second focuses on the application's data-flow. Though these mappings are platform independent and general in nature, they are implemented in a manner specific to the SC synthesizing compiler. To enable debugging, other compilers need to provide similar mappings, customized slightly to the compilation scheme of the compiler. The mappings provided by the SC debug database are reviewed briefly below.

Control-Flow Mappings

L1. **Source Line** $\longleftrightarrow$ **Original Operation.** This mapping allows the debugger to correlate operations in the original data-flow of the program with the line numbers in the source code from which they were derived.

L2. **Original Operation** $\longleftrightarrow$ **Initial Schedule.** Mapping operations to their initial schedule allows the debugger to determine the original order of operations, as described in the source code.

L3. **Original Operation** $\longleftrightarrow$ **Final Operation.** This mapping correlates the original and final operations of the application and provides the debugger with a means of unraveling the optimizations performed by the compiler.

L4. **Final Operation** $\longleftrightarrow$ **Schedule.** This mapping provides the debugger with information about when the final operations will execute in the final, fully optimized circuit.

L5. **Operation** $\longrightarrow$ **Predicate Equation.** This mapping provides the conditions which must be met for the results of an operation to be considered valid.

L6. **Schedule ⟷ Circuit State.** This mapping is used to correlate the logical schedule with the equivalent circuit state.

L7. **Operation ⟶ Breakpoint Unit Programming Data.** This is a special mapping, which provides information specific to a given breakpointing implementation. The information provided by this mapping allows the debugger to program the inserted breakpoint units to stop on a specific cycle, and optionally, only when a certain predicate equation is met.

Data-Flow Mappings

V1. **Source Variable ⟷ SSA Variable.** This mapping allows the debugger to determine from which source variable each SSA variable was derived from.

V2. **Initial SSA Variable ⟷ Final SSA Variable.** This mapping provides information about how the data-flow of the application is changed during the addition, deletion or merging of SSA variables. It allows the debugger to determine where the value for a particular SSA variable is stored in the final circuit.

V3. **SSA Variable ⟷ Operation/Instruction.** Mapping SSA variables to operations allows the debugger to determine when variables are read from and/or written to.

V4. **Variables ⟶ Circuit Element.** This mapping is used to map all internal variables to circuit elements in the final hardware.

V5. **SSA Variable ⟶ Hardware Width.** This mapping provides the final hardware width of each variable, allowing the debugger to properly display variable values to the user.

11.1.3 Additional Capabilities of Hardware Debuggers

With the platform features and mappings described above in place, it is possible to provide observability and controllability of the synthesized circuit in the context of the

original source code. This is a new paradigm, and an important step forward for debugging synthesized circuits because it provides the user with a familiar, intuitive debugging environment.

The hardware debugger is able to provide features typical in software debuggers, including single stepping, breakpointing and watching and setting of variables. In addition, because of the flexibility of FPGAs, the hardware debugger is also able to provide features that are not available in typical software debuggers. This includes the ability to provide two modes of operation regarding the way in which controllability and observability are provided to the user, the limited ability to step execution backward, and the ability to provide more consistent results when setting variable values.

Two Modes of Operation

Because the hardware debugger has access to more of the internal state of a circuit than a software debugger would have of the internal state of a CPU, the hardware debugger can provide more than one view of the executing application. In one view, a software-centric view, the debugger appears much as a typical software debugger: only a single line of code is shown to be active at a time. This view is created using Virtual Sequentialization to make it appear as if the instructions in the program were running sequentially and is a good tool for functionally debugging the circuit. The second view, a hardware-centric view, shows the user information about all operations currently active in the circuit. This would be equivalent to a software debugger showing information about instruction reordering and parallelization which is happening internal to a superscalar CPU. This view can provide added insight for the user into how the compiler has optimized the program.

Stepping Execution Backward

Because of the flexibility inherent in using FPGAs, it is possible to allow a circuit's execution to be run backward. This is enabled by adding checkpointing and rollback circuitry to the synthesized hardware. This hardware is used to allow the circuit to take intermittent checkpoints and return execution to the last checkpoint. This additional circuitry

can be expected to incur a circuit overhead of about 2-6%, in the average case. The ability to move execution backward provides the user of the debugger with added flexibility.

Providing More Consistent Results for Setting of Variables

The addition of the checkpointing and rollback circuitry also allows the hardware debugger to provide more consistent results when setting variable values. In a software debugger, it is typically possible to set the state of a variable at a time when the variable value would not normally change. This can give the user of the debugger a misleading view of the results of changing a variable's value. In the hardware debugger, it is possible to limit the inconsistencies inherent in setting variable values. This is done by returning the circuit state to the cycle on which the variable was last assigned and changing the value immediately afterward. The circuit is then run forward to the point where the process started. In this way, the value of the variable is changed at the point where it was intended to be changed. This approach does not completely eliminate the possibility of misleading behavior, but it does reduce it.

11.2 Future Work

Since source level debugging of synthesized circuits is a new field of research, there are many topics to pursue during future research. A few of those will be discussed in this section, along with some of the issues which will have to be resolved.

11.2.1 Function Calls

The current work focused on source code which did not include function calls. Future work would need to look at the issues involved in debugging code which includes function calls. There are two main ways of implementing function calls in hardware. The first is to pass execution control to the independent function circuitry and return when done. The second is to use a process known as in-lining to insert the behavior of the function directly into the code which made the function call. Each method has its advantages, for example, the first method uses less hardware, and the second allows for more parallelism.

It would be relatively easy to enable debugging for the method that transfers control to the function circuitry. Within the framework used in this dissertation, this method would simply create new hyperblocks for each function, as needed. The input and output to the function would be accomplished through the use of the thread registers. In this way, a function call would be no different to the debugger than any other set of hyperblocks.

In-lining function calls provides a greater challenge for enabling debugging, and at the same time, is the more likely candidate for implementing function calls in a synthesizer. Most of the challenge comes in the form of finding an intuitive way to provide feedback at the source level; allowing the debugger to determine which instructions are operating would be relatively straightforward. This would be accomplished by keeping track of extra information for in-lined code. The debug database would store not only the line number from which each operation was derived, but would also keep track of the line number on which the original function call was invoked. This approach clearly marks each operation in-lined from the function call, even when there are multiple instances of the function in-lined into the main body of code.

11.2.2 Chaining Operations

This dissertation looked at applications in which the operations were mapped one per clock cycle. However, it is also possible (and likely desirable) for the synthesizer to chain operations together such that multiple sequential operations execute in the same clock cycle. This means that it will not always be necessary to register every variable in the application. If a variable is needed only by operations which execute in the same clock cycle, then there is no need to register the value. This creates some interesting issues. First, it means that some variables specified in the source code will not be directly accessible in state elements. Stranger yet, a variable may have a registered and an unregistered version in the hardware.

For variables which are not stored in state elements, it is not possible to directly get the value of these variables using the state reading capability of FPGAs. However, using a hardware interface layer such as JHDL allows the debugger to reconstruct these signals given the state of the variables used to create it. In similar fashion, there is no easy

way to directly set the state of a variable which is not stored in a register. I would suggest two potential ways to overcome this problem. The first is to insert muxes into the circuit in front of any asynchronous signal that the user would like to be able to set. The second is to simulate the effects of changing the variable and setting the state of any registered signal that is affected. Research would be needed to determine if either of these approaches is feasible.

For variables which are used as both registered and unregistered signals, things become even more interesting. In this case, it is possible to set the value of the registered version, but not the unregistered version. This means that simply setting the variable value could affect some operations, but not others. Again, this could likely be overcome with either of the two methods mentioned above. As with above, this really poses no problems to circuit state observation, as the unregistered signal value could easily be reconstructed by JHDL.

11.2.3 Resource Sharing

Another interesting optimization which could be studied is resource sharing. This optimization allows operations to share the same computational unit and allows variables to share the same storage location in the final circuit. Sharing of computational units poses not problems to the current implmentation of the debugger, as the currently executing instructions are determined by the state of the control circuitry. However, sharing storage locations would necessitate the incorporation of additional information into the debug database. In particular, mapping V4:Var-Circuit would require the addition of a temporal aspect. This would be needed so that the debugger knew which variable a particular storage location represented at any give time during execution.

In addition to the changes to the debug database, this optimization would also effect the buffering hardware, since any shared variable storage needs to be buffered to insure that variable values are available to the debugger at the proper times.

11.2.4 Other Optimizations

There are many other compiler optimizations which are not supported in the SC Debugger, which need to be looked at in the future. These include loop unrolling and optimizations which move code into different blocks (interblock code movement). These optimizations can be easily accounted for in the debug database, but can present problems to the debugger.

Loop Unrolling

To support loop unrolling, the debug database must annotate each unrolled instruction such that the debugger knows in which iteration of the loop the operation would execute if no loop unrolling had been done. This information is likely not difficult for the debug database to collect. However, using this information in the debugger can pose interesting problems. In clock step mode, the largest issue is how to present the information to the user; when a loop has been unrolled, multiple instances of the same operation can be active at one time. Also, a single variable on a line of source code can now represent many different variables in the unrolled implementation. A similar problem arises when using SSA, however the solution used for SSA does not work in this case, as it may not always be obvious which value applies to which iteration of the loop.

For source step mode, the problem of presenting the information to the user disappears, however, the process of Virtual Sequentialization is complicated by the fact that a single original operation becomes multiple final operations. Future work would need to determine how to take this into account.

Interblock Code Movement

Interblock code movement occurs when an operation from an original basic block does not remain in the same block as the other operations in the block. This type of optimization can be accounted for in the debug database quite readily by using Mapping L3:OrigOp-FinalOp. However, moving code outside of the original block could complicate the process of Virtual Sequentialization. Future research would need to look at

specific optimizations which move code between blocks and determine how to implement Virtual Sequentialization in the presence of these optimizations. Luckily, this type of optimization has no impact on clock step mode, as the mode simply shows the user what is currently executing in the circuit.

11.2.5 Profiling

Since synthesis tools may in-line all method calls, the typical software approach of reporting total times spent in each method is not necessarily the best approach. A better approach may be to allow the user to select a range of lines in the source code and ask how much time is spent in that range. It may also be possible to report how much execution time is spent on each line. It may actually be possible to get more information from the hardware profile than is possible with a software profiler.

However, the ability to gather this amount of information comes with a price: it will be necessary to add hardware to the circuit to gather the statistics required to generate the information. The good news is that this hardware can be added in parallel to the synthesized circuitry, thus having no effect on the operation of the rest of the circuit[1]. This is an advantage over a software profile which can actually change the profile while gathering the information, making it impossible to get a totally accurate profile.

The goal of implementing profiling is to minimize the amount of information which is needed from the running circuit, thus minimizing the amount of on-chip memory required to store the information. This will mean that much of the profiling information will need to be recreated from the statistics of the running circuit, as well as information in the debug database. For example, for SC-synthesized circuits, it will only be necessary to store the order of hyperblock execution, and the predicate values generated in each execution of a hyperblock. The information in the debug database can then be used to recreate the execution sequence of the circuit.

[1] It is possible, even likely, that the addition of this circuitry will reduce the frequency of the circuit. However, since we can use clock cycles as a metric for the profile, rather than time, this will not affect the results of the profiling.

The research described in this dissertation has studied source level debugging of circuits generated through code synthesis. This work provided the groundwork for allowing a user to debug a circuit in the context of the original source level description of the application, and provided a prototype in the form of the Sea Cucumber Debugger. Providing improved debugging support for synthesized circuits is an important improvement given the increasing use of automated circuit generation tools.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics*, vol. 38, no. 8, 1965.

[2] S. T. Mangelsdorf, R. P. Gratias, R. M. Blumberg, and R. Bhatia, "Functional verification of the HP PA 8000 processor", *Hewlett Packard Journal*, August 1997.

[3] Sun Microsystems, Inc., *Sun Technical Compute Farm (Sun TCF) Whitepaper*, April 2003.

[4] K. Bondalapati, P. C. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler, "DEFACTO: A design environment for adaptive computing technology", in *IPPS/SPDP Workshops*, 1999, pp. 570–578.

[5] J. M. P. Cardoso and H. C. Neto, "Macro-based hardware compilation of java bytecodes into a dynamic reconfigurable computing system", in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, K. L. Pocek and J. M. Arnold, Eds., Napa, CA, 1999, p. n/a, IEEE.

[6] S. Swan, D. Vermeersch, D. Dumlugöl, P. Hardee, T. Hasegawa, A. Rose, M. Coppolla, M. Janssen, T. Grötker, A. Ghosh, and K. Kranen, *Functional Specification for SystemC 2.0*, Open SystemC Initiative, 2.0-p edition, October 2001.

[7] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language", in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2000, p. n/a, IEEE.

[8] Celoxica, *Handel-C Language Reference Manual*, Celoxica Limited, 2001.

[9] G. Snider, B. Shackleford, and R. J. Carter, "Attacking the semantic gap between application programming languages and configurable hardware", in *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2001, ACM, pp. 115–124, ACM Press.

[10] J. L. Tripp, P. A. Jackson, and B. L. Hutchings, "Sea Cucumber: A synthesizing compiler for FPGAs", in *Field-Programmable Logic and Applications*. September 2002, pp. 875–885, Springer.

[11] K. S. Hemmert and B. Hutchings, "Issues in debugging highly parallel FPGA-based applications derived from source code", in *Proceedings Asia and South Pacific Design Automation Conference 2003*, Kitakyushu, Japan, 2003, pp. 483–488.

[12] K. S. Hemmert, J. L. Tripp, B. L. Hutchings, and P. L. Jackson, "Source level debugger for the Sea Cucumber synthesizing compiler", in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2003, pp. 228–237, IEEE.

[13] P. T. Zellweger, *Interactive Source-Level Debugging of Optimized Programs*, PhD thesis, University of California, Berkeley, May 1984, This work is also available as Technical Report CSL-84-5 from Xerox PARC.

[14] P. Pineo and M. Soffa, "Debugging parallelized code using code liberation techniques", in *Proceedings of ACM/ONR SIGPLAN Workshop on Parallel and Distributed Debugging*, May 1991, pp. 103–114.

[15] J. Hennessy, "Symboilic debugging of optimized code", *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 323–344, July 1982.

[16] L. E. Cool, "Debugging VLIW code after instruction scheduling", Master's thesis, Oregon Graduate Institute of Science & Technology, 1992.

[17] M. Copperman, "Debugging optimized code without being misled", *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 387–427, May 1994.

[18] L.-C. Wu, R. Mirani, H. Patil, B. Olsen, and W. mei W. Hwu, "A new framework for debugging globally optimized code", in *SIGPLAN Conference on Programming Language Design and Implementation*, 1999, pp. 181–191.

[19] L.-C. Wu, *Interactive Source-Level Debugging of Optimized Code*, PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[20] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors", in *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp. 266–275.

[21] IEEE, *IEEE Standard VHDL Language Reference Manual*, 2000.

[22] IEEE, *IEEE Standard Hardware Design Language Based on the Verilog Hardware Description Language*, 1995.

[23] Celoxica, *DK1 Design Suite User Manual*, Celoxica Limited, 2002.

[24] Synplicity, Inc, Sunnyvale, CA, *Identify Datasheet*, 2003.

[25] P. S. Graham, *Logical Hardware Debuggers for FPGA-Based Systems*, PhD thesis, Brigham Young University, 2001.

[26] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A CAD suite for high-performance FPGA design", in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, K. L. Pocek and J. M. Arnold, Eds., Napa, CA, April 1999, IEEE Computer Society, p. n/a, IEEE.

[27] P. Graham, B. Hutchings, and B. Nelson, "Improving the FPGA design process through determining and applying logical-to-physical design mappings", in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, B. L. Hutchings, Ed., Napa, April 2000, IEEE Computer Society, pp. 305–306, IEEE Computer Society Press.

[28] Xilinx Corporation, *Virtex Series Configuration Architecture User Guide*, September 27, 2000, XAPP151 (v1.5).

[29] T. B. Wheeler, "Improving design obervability and controllability for functional verification of FPGA-based circuits using design-level scan techniques", Master's thesis, Brigham Young University, 2001.

[30] Xilinx, *System Generator User Guide*, Available at http://www.xilinx.com/products/software/sysgen/app_docs/user_guide.htm.

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form", in *16th Annual ACM Symposium on Priciples of Programming Languages*, 1989, pp. 25–35.

[32] S. S. Munchnick, *Advanced Compiler Design and implementation*, Morgan Kaufmann Publishers, Inc., San Fransisco, California, USA, third edition, 1997.

[33] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante, "Predicated static single assignment", in *IEEE PACT*, 1999, pp. 245–255.

[34] J. Park and M. Schlansker, "On predicated execution", Tech. Rep., Hewlett Packard Laboratories, May 1991, HPL-91-58.

[35] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. W. en, "Conversion of control dependence to data dependence", in *Symposium on Principles of Programming Languages*, 1983, pp. 177–189.

[36] Intel, *Intel Itanium Architecture Software Developer's Manual. Volume 3: Instruction Set Reference*, 2.0 edition, December 2001.

[37] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock", in *25th Annual International Symposium on Microarchitecture*, 1992.

[38] A. W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, 1998.

[39] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.

[40] W. J. Landaker, "Using hardware context-switching to enable a multitasking reconfigurable computer system", Master's thesis, Brigham Young University, 2002.

[41] P. Bellows, B. Schott, and L. Wang, *SLAAC1-V VHDL Guide*, USC Information Sciences Institute—East, Arlingtonm, VA, 2.0.0 edition, July 2002, This document is included with the SLAAC1-V board documentation.

[42] *The Programmable Logic Data Book 2002*, Xilinx Corporation, 2000.

[43] *Virtex-II Platform FPGA Handbook*, Xilinx Corporation, 2000.

[44] Xilinx, *Libraries Guide*, March 2003.

[45] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using design-level scan to improve FPGA design observability and controllability for functional verification", in *Field-Programmable Logic and Applications. Proceedings of the 11th International Workshop, FPL 2001*. August 2001, Lecture Notes in Computer Science, pp. 483–492, Springer Verlag.

[46] S. A. Guccione and D. Levi, "Run-time parameterizable cores", in *Field-Programmable Logic and Applications. Proceedings of the 9th International Workshop, FPL '99*, P. Lysaght, J. Irvine, and R. Hartenstein, Eds., Glasgow, UK, August/September 1999, vol. 1673 of *Lecture Notes in Computer Science*, pp. x–x, Springer-Verlag.

[47] B. Schott, P. Bellows, M. French, and R. Parker, "Applications of adaptive computing systems for signal processing challenges", in *Proc. of the Asia South Pacific Design Automation Conf.*, 2003.

[48] P. Stanford and P. Mancuso, Eds., *EDIF: Electronic Design Interchange Format Version 2 0 0*, Electronic Industries Association, 1989.

[49] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, London, UK, 1985.

[50] S. Oaks and H. Wong, *Java Threads*, O'Reilly & Associates, Inc., 1997.

[51] D. Lea, *Concurrent Programming in Java*, Addison-Wesley Publishing, 2nd edition, 1999.

[52] P. A. Jackson, "Simulation and synthesis of CSP-based inter-thread communication", Master's thesis, Brigham Young University, 2003.

[53] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a Java optimization framework", in *Proceedings of CASCON 1999*, 1999, pp. 125–135.

[54] R. Morgan, *Building an Optimizing Compiler*, pp. 86–90, Digital Press, 1998.

[55] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, chapter 3, p. 133, Morgan Kaufmann Publishers, Inc, second edition, 1998.

[56] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, Addison-Wesley, second edition, 2000.

[57] A.-R. Adl-Tabatabai and T. Gross, "Source-level debugging of scalar optimized code", in *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996, pp. 33–43.

[58] J. P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley Publishing Company, 1993.

[59] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, "Bitwidth cognizant architecture synthesis of custom hardware accelerators", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1355–1370, November 2001.

[60] A. L. Slade, “Designing, debugging, and deploying configurable computing machine-based applications using reconfigurable computing application frameworks”, Master’s thesis, Brigham Young University, 2003.

[61] Brigham Young University, *JHDL User’s Manual*, Available at http://www.jhdl.org/documentation/users_manual/manual.html.