# Issues in Debugging Highly Parallel FPGA-based Applications Derived from Source Code

K. Scott Hemmert and Brad Hutchings
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT, USA
{hemmert,hutch}@ee.byu.edu

**Abstract— Using high-level synthesis tools to map programs written in general-purpose languages to FPGA hardware has grown in popularity and it is becoming necessary to provide comprehensive debugging tools in order to verify the correctness of the synthesized hardware. Currently, post-synthesis debugging is done at the circuit level. This paper discusses the issues, as well as some early results, of creating a source level debugger for hardware synthesized from source code. This study is meant to provide some insight into what needs to be added or built into synthesizing compilers in order to allow debug of a synthesized circuit at the source level, which will provide the programmer with a familiar view of the program being debugged.**

## I. Introduction

With the increasing use of synthesizing compilers which create FPGA circuits from general purpose programming languages [2, 5, 10], computer programmers are becoming more and more involved in the process of creating hardware. While this can have many advantages, it can be difficult for a software engineer without hardware experience to debug a synthesized circuit which does not work properly.

In an ideal world, the synthesized circuit would behave exactly as the program from which it was generated, thus allowing the programmer to debug software rather than the synthesized hardware. However, there are cases where this may not be true. For example, due to the complexity of synthesizers, there is no guarantee that the resulting circuit will behave identically to the software. Also, during the synthesis process, the synthesizer will need to make decisions about the widths of variables (or they will need to be specified by the user). To reduce the size of the synthesized circuit, these widths will typically be smaller in hardware than in software, possibly creating different behavior.

It is also beneficial to debug the actual hardware, as it could potentially be much more efficient. Since the software is being mapped to hardware to greatly improve performance, execution times of the software may be too long to provide efficient debugging.

In order to study how we can facilitate the debugging of synthesized circuitry, we propose to build a hardware debugger that will provide feedback to the programmer using a source-level view. The goal is to provide a feature set which provides the same controllability and observability as that of a typical software debugger. This will allow the programmer to debug the code in a context similar to that in which the code was written. The feature set for the hardware debugger will include the following:

1. **Controllability.** Provide the user control over the state of the running circuit by allowing the user to single-step, set breakpoints, and set the values of variables.

2. **Observability.** Provide the user with views of the state of the program by showing the location of current execution points and allowing the watching of values of variables.

3. **Performance debugging.** Provide the user information to improve the performance of the program, by gathering profiling information from the application executing in hardware and relating it back to the original source code.

As with any software debugger, the effectiveness of the hardware debugger depends on information provided by the compiler. We will refer to this information as the debug database. The issues in creating this database are similar to those involved when mapping code to very long instruction word (VLIW) machines, where you must deal with explicit parallelism, code reordering, optimizations, etc. Although there are a number of similarities between mapping code to hardware and mapping code to VLIW machines, the problems are more pronounced in hardware mapping. For example, a VLIW compiler need only find enough parallelism to keep all the available units in the target CPU busy. When mapping to hardware, the synthesizer is free to find and exploit all available parallelism. This can result in an order of magnitude more parallelism and an accompanying increase in debugging complexity.

Due to the complexities of debugging optimized code, many people advocate turning off optimizations when debugging, though there have been a number of studies done on how to debug optimized code [3, 4, 13]. This work extends many of the ideas proposed in these previous works to apply to the task

at hand. Specifically, we are introducing two modes of operation for the debugger to debug circuits generated from general purpose programming languages. The first mode provides "truthful behavior" [14] and is called clock step mode. The second mode of operation is called source step mode and provides "expected behavior" to the user by using a novel buffering approach. These approaches will be discussed in greater detail in Section IV.

Whereas some approaches allow debug for only a subset of optimizations, it is important for this work that the user be allowed to debug the final, fully optimized circuit. This guarantees that the user doesn't debug an unoptimized circuit, only to find that the optimized circuit does not operate correctly. Debugging the fully optimized circuit will also provide the programmer with insight into how the synthesizer is mapping the code, allowing the code to be rewritten in a more efficient manner.

This paper will discuss the issues involved in creating both a hardware debug database and a hardware source-level debugger for circuits derived from general purpose programming languages. We will begin by giving some necessary background on synthesizing compilers. We will then talk specifically about the information required in the hardware debug database, and how this information is used by the hardware debugger. We will also discuss some general issues involved with debugging hardware at the source level.

## II. Background

To provide ourselves with a well defined scope for the project, we chose to concentrate our work on synthesizing compilers that use predicated static single assignment (PSSA) [1] to find exploitable parallelism. This work will also apply to those compilers which use only predication or static single assignment (SSA). In order to verify our results, we chose to use the Sea Cucumber [11] compiler, as it is representative of synthesizing compilers available today and because we have access to the source code for the compiler.

Sea Cucumber reads in Java bytecode and generates an optimized circuit to implement the behavior of the bytecode. A graphical representation of the Sea Cucumber framework is shown in Fig. 1. During the analysis of the bytecode, all references to variables are made unique by applying SSA techniques. This is done by creating a new version of the variable each time it is assigned a new value. SSA reduces the number of dependencies in the final program which allows for more instruction reordering. The output of the bytecode analyzer is a control flow graph where each node represents a basic block of the program.

As the control flow graph is converted to the internal data structure, the compiler will create hyperblocks [9] from the basic blocks. This is accomplished by applying predication to the instructions in each basic block. Predication makes it possible to compute all branches of a conditional statement in parallel. The result is selected later, after the condition has been computed. This is done by adding a predicate equation

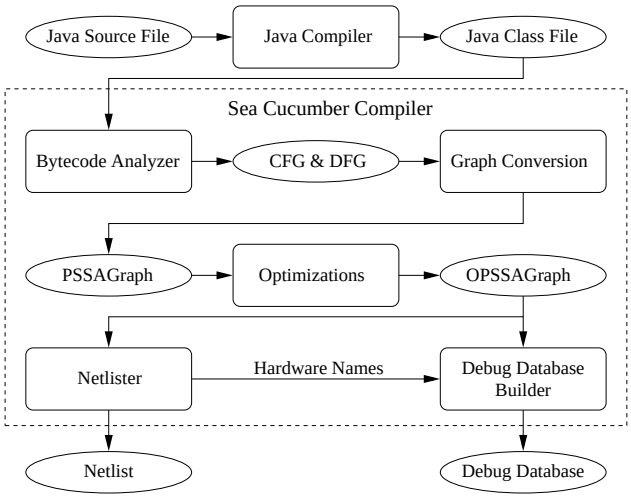


Fig. 1. Overview of operations performed by Sea Cucumber during the synthesis process, including operations we have added to create the debug database. Oval nodes represent data files or formats; rectangular nodes represent operations or tools.

to each instruction in the hyperblock. The equation gives the conditions that must be met for the result of that operation to be committed. Using predication allows hyperblocks to be a much larger code grouping than a basic block. This allows the compiler to do more code reordering, thus allowing more parallel execution.

After the hyperblocks are formed, the compiler will then apply several optimizations to improve efficiency and will generate a netlist of the synthesized circuit. We have also added the ability to the compiler to create a debug database for the circuit. The general contents of this database are discussed in the next section.

## III. Hardware Debug Database

The information required in a hardware debug database parallels that required for software debugging and can be put into two general groups: variable table and line number table. While the information needed in the databases is similar, the hardware database generally requires more information than its software counterpart. The following sections will compare the information in a typical software symbol table with that needed for a hardware debug database.

### *A. Line Number Table*

For a software debugger, the line number table is used to map program counter values or offsets in the program to the line numbers in the source file responsible for the creation of the operation stored at each program offset. In this case, the process of enumerating this information is quite straightforward (at least for unoptimized code). Because of the distribution of computation in a synthesized circuit, recording this type of information is much less direct.

As no program counter exists in the hardware, the hardware debug database is required to store information that shows the relationship between the state of the control circuitry and the line numbers of the source file. The way this relationship is established can be quite different depending on the synthesizer. There is also another related problem in hardware that does not exist in software. Because the hardware representation is not simply a list of instructions to process as it is in software, the scheduling information for the instructions is not given explicitly. This information much be computed and inserted into the debug database.

For circuits which utilize predication, there is another issue to tackle. With predication, instructions can be executed and later invalidated. If the debugger wishes to differentiate between instructions whose predicate equations are met or those whose are not, then it will be necessary for the debug database to list the predicate equation for each operation.

While working on the debug database for Sea Cucumber generated circuits, we found it easiest to store the line number, schedule and predicate information in data structures that would be comparable to assembly code in software. The data structure essentially contains a list of all the hyperblocks along with the operations within each hyperblock. Stored with each operation is its line number, schedule information and predicate equation. For Sea Cucumber, the scheduling information is simply a reference to the state number during which an instruction executes within its hyperblock.

### *B. Variable Table*

For software compilers the variable table contains a list of the names of all variables, along with their memory locations and scoping information. This allows the debugger to find and set the value of each variable. For hardware debuggers, the variable table is slightly different; rather than mapping variables to memory locations, the hardware variable table maps the variables to locations (circuit elements) in hardware. The debugger will also need to know to which type of circuit element the variable is mapped. For example, if the variable is not mapped to a state element (register), then it is possible that the debugger will not be able to set its state. It will also not be possible to directly read the state of the variable, but it may be possible that the value can be computed given the values of the variables used to create it.

For synthesizers which use SSA, it is also necessary to have a list of each version of each variable, along with information about the sections of code for which each version is valid. It is also necessary for the variable table to contain information about variables that are not explicitly found in the source file. This is the case with the variables used for predication. These variables are the boolean values generated from conditional statements. The values of these variables are not typically shown in the debugger, but the debugger will need to know their values in order to correctly display other information about the running program.

For the Sea Cucumber debugger, the debug database contains a listing of all versions of all variables in the final circuit, whether or not they are explicitly found in the source file. The names of the variables from the original source file are extracted from the local variable table in the Java class file from which the circuit is synthesized. As the compiler creates the variables used for predication, they are given unique names generated by the compiler. The name of the circuit element which holds the value of the variable in the final circuit is stored long with each variable. This allows the debugger to extract the values of the variables at runtime.

## IV. Hardware Debugger

The way source code is mapped to hardware greatly affects the ability of the debugger to support the features listed in Section I. The main issues arise because of the compiler's many degrees of freedom in creating custom hardware. This ranges from the ability to exploit all available parallelism to the increased clock control gained in mapping to FPGAs.

Because of the increased clock control of FPGA circuits over that of a general purpose CPU, it is possible to define a single-step in two ways: clock stepping or source stepping. The choice of definitions plays an important role in how the feature set of the debugger is implemented, and each has advantages and disadvantages. We will first define each of the associated stepping modes and then describe the general hardware issues with the implementation of each of the debugger features, as well as issues specific to each stepping mode.

**Source Stepping** Source stepping is meant to imitate the single-stepping found in a software debugger. It allows the user to see the execution move sequentially through each line in the source code, even though the circuit is actually executing many instructions in parallel. This is a much more familiar view for a software engineer, but poses some interesting problems due to the parallel nature of the hardware as compared to the sequential nature of the source code. These issues are discussed in the following sections.

**Clock Stepping** Clock stepping defines a single-step as a single cycle of the clock running the circuit. In this mode, the user would be shown all the lines of code that are executing on any given clock cycle. This has the advantage of simplicity, but the disadvantage of being unfamiliar to the programmer. Another advantage of this method is that it gives a more accurate view and allows more insight into the running circuit.

### *A. Single-Stepping*

Both methods of single-stepping depend on the availability of certain features in the target platform. The ability to precisely control the execution of the circuit requires control of the clock from both the host and from the circuit. This allows the debugger to do both single-stepping (host controlled) and breakpointing (circuit controlled). If the platform does not support clock control, then the synthesizer would have to add circuitry in order to provide this feature.

Whereas single-stepping in clock step mode is very straightforward (the debugger simply advances the hardware clock one cycle for each single-step), single-stepping in source step mode is more complex. In order to single-step in source step mode, the debugger needs to refer to the debug database to find what operation represents the next line of source code. The debugger must then advance the clock until this instruction is executed.

When advancing the clock multiple cycles in order to execute the next line of code, the debugger will need to buffer the state of the circuit, because future instructions in the source code may actually execute during these clock cycles. Thus, if the next instruction was already executed, stepping to it may involving looking back into the state buffer, rather than advancing the clock.

### *B. Breakpointing*

In order to be consistent, the definition of a breakpoint changes depending on which stepping mode is chosen. In clock stepping mode, a breakpoint is triggered when the operation synthesized from the breakpointed line is executed. In the case of a line resulting in multiple operations in the final circuit, the operation scheduled earliest is used to trigger the breakpoint.

The most important issue in this mode arises when the compiler uses predication. Because instructions are predicated, many instructions will be executed even though they would not be executed in a strictly sequential execution. In the compiled circuit, the results of these instructions are simply never committed if their predicates are not satisfied. Execution should only stop if the predicate equation for that operation has been satisfied. However, the predicate may be scheduled to be executed after the actual instruction is executed. In this case, it may be impossible to stop execution exactly on the breakpointed line, however, the debugger would be able to tell the user exactly how many cycles late execution will stop.

In the source stepping mode, a breakpoint would be triggered once all previous instructions (in the source file) of the breakpointed instruction have executed. If predication is used all predicates used by any of these instructions would also need to be computed before the breakpoint is triggered. This is consistent with the results you would get if you single-stepped the circuit to the breakpoint.

Because of the necessity to buffer information while stepping, it is not enough to allow the hardware to free run to the breakpointed instruction. If this occurs, then the debugger will not have the information buffered to reconstruct information for reordered instructions. Of course, if the circuit were single-stepped from this point, the buffer could be filled, and eventually things would operate as usual. There are two ways to solve this problem. In the first approach, the debugger finds an earlier instruction on which to break. The instruction is selected such that it allows the debugger to single-step to the actual breakpoint and fill the state buffer. The second approach is to store this information in the circuit by adding extra circuitry. However, this solution would necessarily limit the visibility into the circuit because of storage issues.

### *C. Setting Values of Variables*

We intend to target the Xilinx Virtex and Virtex II FPGAs. These devices will allow us to use a method described in [8] to read and set the values of variables. This method uses readback to get the state of the circuit, and modifies the configuration bitstream for the FPGA to change the set/reset state of the flip-flops in the design to force all registers to power up with a specific state. This would allow us to change the value of any variable. If a synthesizer targets a platform that does not support these features, then it is also possible for the synthesizer to add a scan chain to the design in order to read and write circuit state [12].

One other concern of mapping to hardware is the fact that the compiler is free to map variables to a variety of circuit elements. For example, if a variable is not mapped to a state element, then its state cannot be set. It is also possible that variables will have multiple versions in the hardware, as is the case when using SSA. The debugger will then need to determine which versions of the variable need to be altered.

Another problem arises when using source stepping: A user may set the value of a variable thinking it will affect the outcome of a future computation, when in fact that computation has already taken place because the compiler scheduled the instructions out of order. The debugger would need to be able to warn the user of this case. Another option is to use the buffered state data to "roll-back" the clock, set the value of the variable and run the clock forward again. It is not clear at this time how difficult this would be to accomplish.

### *D. Location of current execution points*

When using clock step mode, the amount of parallelism that is exploited by the synthesizer means that there are usually a large number of operations executing in parallel. Couple this with the fact that the instructions have possibly been scheduled out of order, and it often becomes impossible to locate a single point of execution in the source code.

Predication complicates this even further. Because of predication, it is possible for multiple branches of a conditional statement to be active at the same time. As the predicate equations for these instructions are calculated, the circuit will know which values to commit for future use. However, these equations may not be fully computed until after many, or all, of the instructions on a given branch have been executed. This makes it very difficult to convey to the user where to find the current execution point. In this case the best thing to do is simply show the user all of the instructions which are currently executing.

### *E. Watching Values of Variables*

While in clock stepping mode, watching values of variables is complicated by the use of SSA. The fact that many versions

of a variable exist makes it necessary to force the user to either specify which version of the variable to watch, or it requires that the debugger determine which of the versions of that variable is currently the correct one, given the current execution points in the program.

Predication will, of course, also play a role in watching variables. The debugger will need to take into account the predicate equation for each variable[1]. If the equation for a variable is not satisfied, then the debugger need not consider that version as one of the possible values of a variable. Again, this is complicated by the fact that the predicate for a variable could be calculated after the assignment to the variable. In this case, the debugger may have to delay reporting the change of value until the predicates are computed.

While in source stepping mode, the debugger does not have these problems. Because the debugger is making it appear that there is a single point of execution, it is well defined which version of a variable is valid at any time, and the predicates are guaranteed to be computed. However, the debugger will need to determine if the value in the current state snapshot is valid or if it needs to look in the buffered state data to find this value.

### *F. Profiling*

Since synthesis tools commonly inline 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 other circuitry, thus having no effect on the operation of the rest of the circuit[2]. 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 profiling will be to minimize the amount of information needed from the running circuit to create the profile. This will, in turn, minimize the amount of on-chip memory required to store the information. Because we will be minimizing the amount of data received from the circuit, the details of the profile will need to be computed in software from the statistics gathered from the running circuit.

[1]The predicate equation of a variable is simply the predicate equation of the instruction which assigns a value to that variable. Because of SSA, each version of a variable is guaranteed to be set only once.

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

## V. Methodology

The Sea Cucumber compiler is still in development, but is able to analyze a subset of Java programs and create circuits to implement the behavior described in the programs. The output of our modified version of Sea Cucumber is a netlist of the synthesized circuit and the hardware debug database required by the debugger. The debugger uses JHDL [7] to simulate the synthesized circuit at the gate level. The debugger communicates with the simulator to control clock stepping and to get values of key circuit elements. These values are then processed using the information in the debug database to present the state of the circuit to the user in the source-level view. Since JHDL simulates at the gate level, this is equivalent to executing the actual circuit in an FPGA. Because there has been a considerable amount of research done which has resulted in JHDL having the built-in ability to communicate directly with circuits executing in FPGAs [6], we plan to use JHDL to act as the communication interface between the debugger and the synthesized circuit running on the FPGA.

Using JHDL as either the simulation layer or the hardware communication layer also provides us with another useful debugging tool: circuit visualization. Since JHDL provides a structural view of the synthesized circuit, it is possible to get information both at the source level and at the circuit level. This has proven quite useful to us while writing and verifying the debugger. This type of functionality could also prove a boon to programmers using the Sea Cucumber system. It would allow them to gain some insight into how the synthesizer works, as well as provide additional information at debug time (assuming the programmers were also versed in hardware design).

## VI. Preliminary Results

In order to verify the correctness of the information in the hardware debug database, we used the approach described in the previous section to create a plug-in to JHDL which provides a source-level view of a simulating circuit. We have prototyped the location of execution points, as well as the watching of variable values, using clock stepping mode. A screen capture of the prototype tool is shown in Fig. 2. The initial results have been quite promising.

The tool extracts the state of the running circuit from the JHDL simulation and highlights all of the currently running operations. This is done by looking at the state bit which indicates the idle state for each hyperblock and determining which hyperblock is not currently idle; this is the active hyperblock. The debugger then looks at all of the state bits for the active hyperblock to determine its current state. It then refers to the debug database to determine which instructions are executed during that state, along with their corresponding line numbers in the source code. These lines are then highlighted in the source level view.

The debugger also shows the current values of all variables found in the source code. It does this by looking at the currently executing instructions to determine which version of the

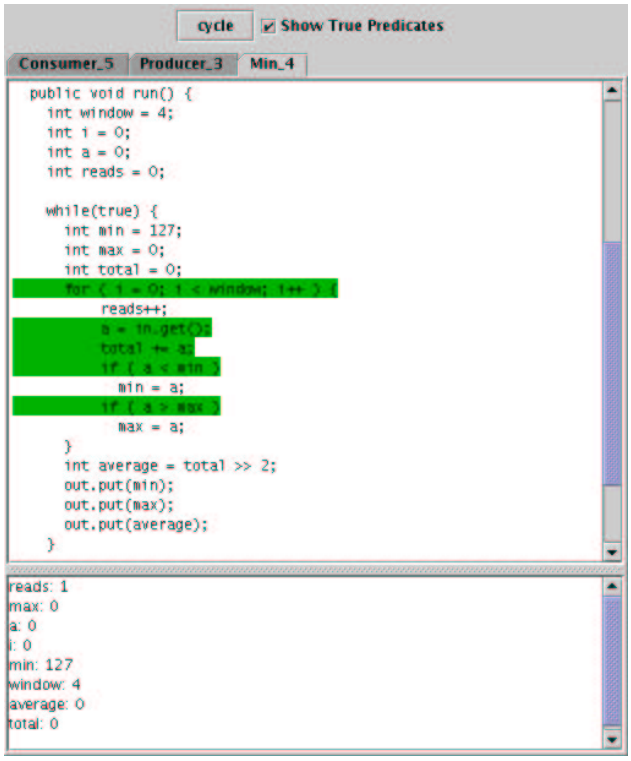


Fig. 2. Screen shot of the prototype debugger for Sea Cucumber, showing the location of current execution points and variable values.

variable is currently valid. If more than one version is currently valid, all the valid values are displayed. Optionally, the tool can also show the user the value of each variable on a line by line basis. We have found this to be a very effective way of interpreting the circuit given the complex nature of circuits generated using PSSA.

## VII. Conclusions and Future Work

We have had some promising early results with the work on the source-level debugger for synthesized hardware. With our prototype debugger, We have been able to demonstrate that it is possible to identify the current execution points of a running circuit, as well as watch the values of variables. This also demonstrates the ability of the compiler to build the hardware debug database.

We will continue to implement the remaining features in the debugger using clock step mode. This will allow us to verify the information that needs to be present in the hardware debug database. After we have verified the information in the debug database and the features of the debugger, we will re-verify the work in hardware by downloading the circuits to FPGAs.

After we have created the full debugger using clock step mode, we will extend the work to include source stepping mode. This will allow us to study how the buffering will need to work in order to reconstruct circuit state for out of order execution. We will also look at the possibility of "rolling back" the clock to support variable setting in the presence of reordered instructions.

## References

[1] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated static single assignment. In *IEEE PACT*, pages 245–255, 1999.

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

[3] Lyle Edward Cool. Debugging vliw code after instruction scheduling. Master's thesis, Oregon Graduate Institute of Science & Technology, 1992.

[4] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, May 1994.

[5] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, page n/a, Napa, CA, 2000. IEEE.

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

[7] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A cad suite for high-performance fpga design. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, page n/a, Napa, CA, April 1999. IEEE Computer Society, IEEE.

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

[9] 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.

[10] Stuart Swan, Dirk Vermeersch, Dündar Dumlugöl, Peter Hardee, Takashi Hasegawa, Adam Rose, Marcello Coppolla, Martin Janssen, Thorsten Grötker, Abhijit Ghosh, and Kevin Kranen. *Functional Specification for SystemC 2.0*. Open SystemC Initiative, 2.0-p edition, October 2001.

[11] Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings. Sea cucumber: A synthesizing compiler for fpgas. In Manfred Glesner, Peter Zopf, and Michel Renovell, editors, *Field-Programmable Logic and Applications*, pages 875–885. Springer, September 2002.

[12] Timothy Brian 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.

[13] Le-Chun Wu, Rajiv Mirani, Harish Patil, Bruce Olsen, and Wen mei W. Hwu. A new framework for debugging globally optimized code. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 181–191. ACM Press, 1999.

[14] Polle 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.