Source-Level Debugging for FPGA High-Level Synthesis

Nazanin Calagar, Stephen D. Brown, Jason H. Anderson
Dept. of Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
Email: legup@eecg.toronto.edu

***Abstract*—We describe a source-level debugging framework for FPGA high-level synthesis (HLS) that offers `gdb`-like step, break, and data inspection functionality for an HLS-generated hardware circuit. With the proposed framework, the user can inspect the values of logic signals in the hardware from the `C` source code perspective. The logic signal values come from one of two sources: 1) a logic simulation of the RTL, or 2) an actual execution of the hardware on an FPGA. In addition to the software-like ecosystem for FPGA HLS debugging, the framework provides the user with insight on the RTL produced by the HLS tool for each `C` statement, and permits concurrent hardware and software debugging to discover the first point at which any logic signal in the hardware mismatches with its corresponding variable in software.**

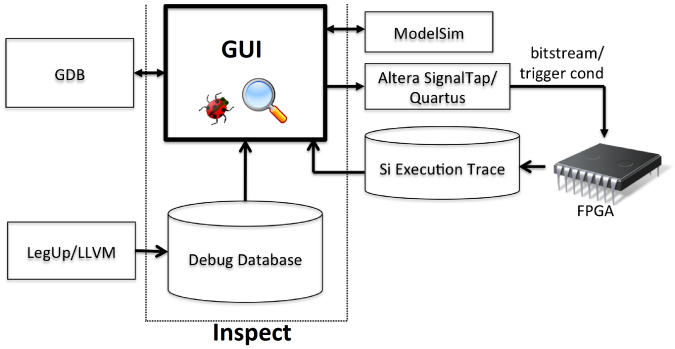

Fig. 1. Components of Inspect debug framework.

I. Introduction

High-level synthesis (HLS) raises the level of abstraction for hardware design by allowing software methodologies to be used. Implementing computations in hardware typically provides speed and energy benefits vs. a software implementation, and the value proposition of HLS is to bring such benefits to two types of users: 1) hardware engineers who use HLS to increase engineering productivity, and 2) software engineers with a limited (or no) knowledge of hardware design. While tremendous strides have been made in the capabilities of HLS tools to generate a functionally correct circuit for a given software program (e.g. [21], [9], [7], [18], [14]), a crucial aspect of the normal design process has been sorely lacking in HLS flows: debugging methodologies. Given a bug in the HLS-generated hardware or its integration with a surrounding system, the user is forced into HW-debugging methodologies – logic simulation and manual inspection of waveforms. Thus, debugging HLS hardware is virtually impossible for users without hardware skills. Moreover, we argue that even for HW design experts, debugging tens of thousands of lines of auto-generated RTL is undoubtedly a daunting task. A SW-like debugging framework is therefore needed for HLS to gain broader uptake by both the HW and SW communities. This paper contributes just such a framework.

Our debugging framework, called *Inspect*, offers the user a *software* perspective to debugging HLS-generated *hardware*. Through a GUI, the user is able to step through the `C` source code, set break points, inspect/watch variables, and so on, with the run-time-specific "program" data (values of variables) being retrieved from an execution of the hardware. Inspect supports two modes for hardware execution: 1) simulation of the RTL produced by the HLS tool, and 2) execution of the actual hardware on an FPGA – silicon debug. To realize the first mode, Inspect interacts with a concurrently running logic simulator (ModelSim) to step the hardware cycle by cycle, and extract the values of logic signals. In this mode, the state of *any* variable (logic signal) can be inspected by the user. The second mode uses Altera's Signal Tap II [6] logic analyzer to extract the values of signals from the circuit's execution in silicon. In the silicon debug mode, the limited number of pins, memory and interconnect on the FPGA necessitates that the user pre-selects signals of potential interest. The silicon mode permits the analysis of bugs that are "invisible" at the RTL level, for example, timing-related bugs, transient run-time errors (e.g. SEUs), bugs in the interface between an HLS-generated HW module and other modules in the system, or even bugs in the vendor's RTL/logic synthesis, mapping, placement and routing tools.

Aside from software-style hardware debugging, Inspect provides additional debugging capabilities that are unique to the HLS context. The first is a feature we believe will be useful to HW engineers: the ability to view the relationship between the `C` and the RTL generated by HLS. Stepping through the `C` source, the framework highlights the corresponding lines of Verilog, providing visibility into the amount of instruction-level parallelism extracted by HLS. This feature is made possible through Inspect's *debug database*, which is generated during HLS, and contains the relationship between three "views" of the program/circuit at different levels of abstraction: 1) the `C` source, 2) the representation within the compiler, and 3) Verilog HDL. A second feature is automated SW/HW discrepancy detection, wherein Inspect communicates with concurrently running `gdb` and ModelSim processes on the same machine to identify, at run-time, the first point at which a software execution of the program mismatches with the hardware execution. This feature permits quick pinpointing

of bugs in specific computed values, rather than waiting for errors to propagate to circuit outputs. Lastly, Inspect provides automated discrepancy detection between the RTL simulation of a circuit and its implementation in silicon, with a correlation back to the original C source.

Inspect is implemented using the LegUp open-source HLS tool developed at the University of Toronto [7], which itself is built within the LLVM compiler framework [16]. The debugger presently supports the Altera Cyclone IV FPGA family [5]. We demonstrate the efficacy of the tool in two ways: 1) a case study illustrating Inspect's features and the manner in which a user interacts with the tool, and 2) using a set of benchmark circuits where we have manually inserted bugs into the RTL produced by LegUp HLS. We show that the automated discrepancy detection permits bugs to be discovered in significantly fewer cycles than if they must propagate to circuit outputs. Our tool is also capable of detecting bugs that affect internal signals that may not change the circuit outputs for the given test vectors, and also bugs that cause the HW to execute in an "infinite loop". Our tool is open source and will be incorporated into the next major release of the LegUp framework. Fig. 1 gives an overview of the components of the Inspect debug framework.

The remainder of the paper is organized as follows: Section II describes related work on debugging, the LLVM compiler framework and LegUp HLS. Section III introduces the debug database, which we implemented using MySQL [17]. Section IV describes Inspect framework features in detail and its implementation using a case study. An experimental study is described in Section V. Conclusions and suggestions for future work appear in Section VI.

II. Background

Debugging for HLS is a relatively new area of research, as HLS has only recently received widespread attention and use. Moreover, the first users of HLS have been engineers with HW design skills, making hardware debugging methods possible, albeit undesirable to be applied on auto-generated HLS designs. As adoption of HLS spreads to those without hardware skills, debugging strategies are needed. Testing only the software execution of a program (using a processor) is inadequate, as many different types of bugs may occur in its hardware implementation.

The work of Hemmert et al. on the Sea Cucumber HLS Debugger most closely resembles our own [12], [19]. It provided source-level debugging capability for hardware designs in the context of the original Java source. However, the tool had several limitations, including no support for function calls, or shared hardware resources. Moreover, the Sea Cucumber project was halted in the early 2000s and the debugger was never released publicly. Our framework overcomes these limits, and incorporates new features, such as the automated HW/SW discrepancy detection. Curreri [10] has an assertion-based verification framework for HLS. However, Inspect debugs the entire program as is usually done in software development.

Among commercial tools, Vivado HLS from Xilinx [1] separates the HLS verification flow into two separate steps: the C program and the RTL, therefore lacking a link between the two abstractions and the live discrepancy check feature that our tool offers. The CyberWorkBench HLS tool from NEC [20] offers a GUI similar to Inspect that allows cross-probing between the C and RTL, and co-simulation of the C and a SystemC model of the HW. Calypto Catapult HLS [2] produces a cycle-accurate SystemC model of the generated hardware to ease verification, but likewise lacks automatic RTL HW/SW mismatch checking. Automated testbench generation for the RTL (with vectors extracted from the C) is also supported by the commercial offerings.

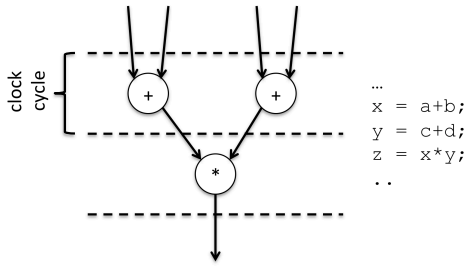

Fig. 2. LegUp scheduler output for a snippet of C.

For debugging in silicon, Xilinx ChipScope Pro [4] and Altera SignalTap II [6] automatically insert trace buffers and trigger circuitry to allow a pre-selected set of signals to be visible in a silicon run. Certus from Mentor Graphics [3] offers a similar functionality, with the ability to observe a greater number of signals. The Inspect framework uses SignalTap II for the silicon debugging mode.

There is currently no publicly available open-source framework to assist with debugging HLS hardware. We aspire to fill this gap in the community and enable a variety of new research in HLS debugging, HW visualization and optimization.

A. The LLVM Framework

LLVM is an open-source compiler framework widely used in industry and academia. LLVM's front-end, *clang*, parses and converts a C or C++ program into LLVM's intermediate representation (IR). The IR closely resembles assembly code, wherein the computations and control flow of the program are represented as simple instructions, e.g. shift, multiply, add, branch, and so on. Following conversion to the IR, the program is optimized using a wide variety of optimization passes. Finally, LLVM's back-end is invoked to generate machine code for the target processor.

B. LegUp High-Level Synthesis

LegUp HLS is implemented as back-end passes in LLVM's framework, accepting LLVM IR as input and producing Verilog HDL code as output. LegUp HLS performs several tasks: allocation, scheduling, binding and Verilog generation. The allocation step makes top-level decisions about the hardware composition, deciding the number of functional units to be used of each type. The scheduling step then assigns the computations and control flow instructions in the LLVM IR into specific clock cycles, thereby defining a finite state machine for the hardware. LegUp's scheduler formulates the problem mathematically, as a linear program [8]. Fig. 2 shows an example of scheduler output for an input C snippet, where

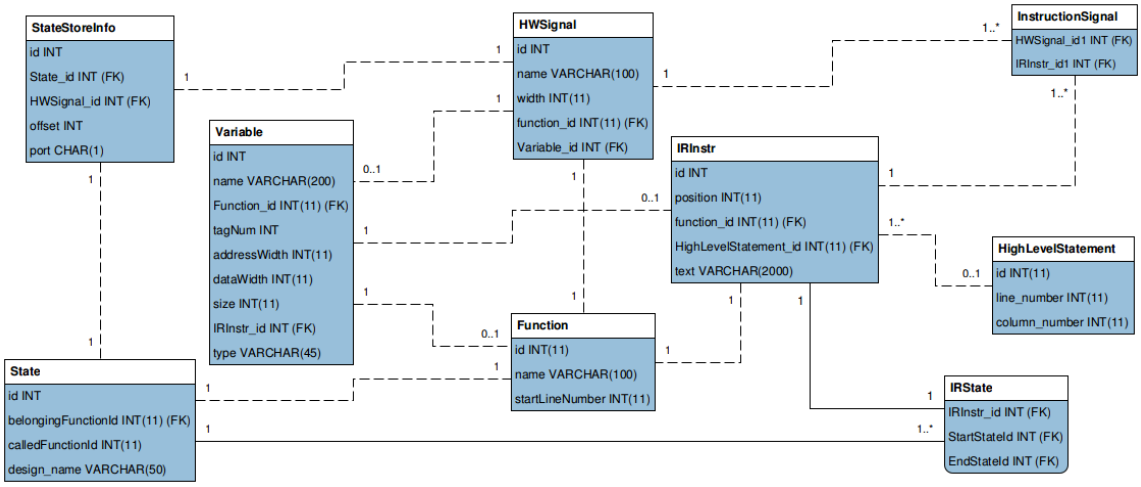

Fig. 3. Simplified Inspect debug database schema. Figure is auto-generated by MySQL.

nodes at the same level of the graph are scheduled in the same clock cycle. We include Fig. 2 to point out that, provided data dependency constraints are met, LegUp can schedule multiple computations in parallel in a cycle – an issue that must be managed in our HLS debugger. Thus, in HLS hardware there may be multiple C statements that are "active" in a given clock cycle. In the figure, two additions occur in parallel and in general a number of operations of different types may occur concurrently. Following scheduling, the binding step decides which specific hardware unit of a given type is to execute each operation in the IR. For a more detailed treatment of LegUp HLS, the interested reader is referred to [7].

III. Inspect Debug Database

Inspect's debug database is the key "information center" that Inspect uses to allow hardware to be debugged and analyzed from the C source perspective. The database contains the relationships between three different views of the program: the C source (software), the LLVM IR (low-level software), and the Verilog (hardware). Using the database, we can find, for each line of C code, the corresponding set of LLVM IR instructions, and likewise, the corresponding computations and FSM states in the hardware. A simplified view of the database schema is shown in Fig. 3 – the figure is automatically generated by the database we used, MySQL [17] – an open-source commercial grade DBMS. Each box represents a data record (many internal fields are not shown for clarity), and edges represent relationships between corresponding records. Note that edge ends have labels that convey 0-to-many relationships. For example, the edge between the HW signal record (HWSignal box) and the variable record (Variable box), has a "1" on one end of the edge, and "0..1" on the other end of the edge. The meaning of these labels indicates that every hardware signal is associated with 0 or 1 variable in the C code – there are many signals in the HW that have no corresponding C variable, hence the possibility of "0".

We highlight key information represented in the database:

1) **High-level C statements:** We track their line and column number in the C file. With this information, Inspect can operate with statements rather than lines. This is useful in situations where either one source line contains more than one C statement (e.g. for loops) or the input source file is not well formatted.

2) **IR instructions:** We store relevant information for LLVM IR instructions including their function, instruction's text, and position (order) in the function. One IR instruction may also have a link to the high-level statement it belongs to. Though, there are some IR instructions that do not have any link to a high-level statement due to either compiler optimizations or a lack of debug metadata. A high-level statement can link to more than one IR instruction creating a one-to-many relationship.

3) **States:** Contains information about LegUp's FSM, including the IR instructions scheduled in each state and the sets of signals whose values may be affected. In LegUp, IR instructions may take one or more cycles to execute, therefore each IR instruction has links to an IRState record defining its start and end states (startStateId and endStateId fields). LegUp states that handle `store` instructions have more information to preserve: 1) the HW signal that corresponds to the variable being updated and its byte offset, and 2) the memory port in which the value is being read/written from/to. This relationship is modeled in StateStoreInfo records.

4) **Functions:** We track the names and file line numbers of all C functions compiled to hardware. Most of the Inspect database entities including variables, IR instructions and HW signals are linked to function records, implying the function they belong to. High-level statements are also linked to functions through their relationship to IR instructions. Each state also has links to a function record, specifying the function in which the state is defined (belongingFunctionId) and the function that the state may call, if it is a call state (calledFunctionId).

5) **HW signals:** We store information about the hardware

design's wires and registers. The information includes the signal name, bitwidth, parent function, and so on. LegUp generates one or more signals for each IR instruction, reflected in InstructionSignal records.

6) **Variables:** Contains necessary information about RAM modules instantiated by LegUp, as well as LLVM IR variables. We store the address width, data width, tag number, size (# of elements), as well as higher-level information on data types. Arrays and structures are supported by Inspect. Variables have three types of relationships with other entities: 1) a link to a function record, 2) a link to HW signal record, and 3) a link to an IR instruction record.

The debug database is produced automatically by Inspect during the HLS flow. Specifically, we altered LegUp so that during its Verilog-generation step, when the HW is completely specified, we populate the contents of the database.

IV. The Inspect Debugger / Case Study

A. Overview

We illustrate the features of Inspect through a case study. Fig. 4 shows a `C` program that counts the number of primes in a 3-dimensional array of `int`. Lines 1-3 declare and initialize the 3D array as a global variable. Lines 4-16 contain a function, `isPrime`, that determines if an `int` type parameter `num` is a prime, returning `0` or `1` accordingly. The `main` function in lines 17-onwards walks through each element of the array and calls the `isPrime` function for each, accumulating the returned values into `result`, which is ultimately returned by the program.

Inspect offers software-engineer-friendly debugging capabilities for HLS-generated hardware, as well as offering additional capabilities for those with hardware expertise. Regarding the first set of capabilities, with Inspect, the user is able to step and execute a program's `C` source code, while behind the scenes (and transparent to the user), the program's hardware implementation is executed. Inspect does not modify the generated Verilog code and works with the exact design generated by LegUp. As the user inspects the values of program variables at run-time, the values are extracted from the corresponding logic signals in the hardware execution. This functionality in itself presents a significant challenge, namely, synchronizing between the software and hardware perspectives. The execution granularity for a software debugger is typically a line of source code; whereas, the execution granularity of hardware is a clock cycle. Broadly speaking there is a many-to-many relationship between lines of `C` source and hardware clock cycles. What this means is that the computations in a single line of program code are frequently scheduled across several different clock cycles, and moreover, a given clock cycle may perform computations that arise from multiple lines of program code. Illustrating such relationships to the user in a coherent and useful manner is a significant hurdle. We solve the problem using multiple GUI windows, one showing the `C` statement being executed, one showing the corresponding LLVM IR, and one showing the corresponding Verilog HDL. From the user's perspective, as the `C` is executed line-by-line, the corresponding lines of IR are highlighted in the adjacent window and the relevant Verilog is automatically shown in the third window.

```
1 : #include <stdio.h>
2 : /**/
3 : int array[2][2][3] = {{{1,2,3},{4,5,6}}, ...
4 : int isPrime (int num)
5 : {
6 :   int i, test;
7 :   test = 1;
8 :   if (num % 2 == 0)
9 :     test = 0;
10:   else {
11:     for (i = 2 ; i < num; i++)
12:         if (num % i == 0)
13:           test = 0;
14:   }
15:   return test;
16: }
17: int main (void)
18: {
19:   int a, b, c, number, result;
20:   result = 0;
21:   for (a = 0; a < 2; a++) {
22:     for (b = 0; b < 2; b++) {
23:       for (c = 0; c < 3; c++) {
24:         number = array[a][b][c];
25:         result += isPrime(number);
26:       }
27:     }
28:   }
29:   ...
```

Fig. 4. `C` code for case study – finding the primes in a 3D array.

Labels ①, ②, and ③ in Fig. 5 illustrate the three program/circuit views and code highlighting in an active debugging session with Inspect. Observe that, since LegUp directly synthesizes the LLVM IR, the FSM state for each IR instruction is displayed in the IR window, giving insight into the hardware, specifically the scheduling results of HLS. In the debug session, the HW is in FSM state #13. Observe in the panel labeled ③ that the hardware is currently performing an integer comparison – the line of Verilog is shown. Highlighting in the panels labeled ① and ② illustrate that presently, the HW is computing the comparison in the inner-most loop.

Two modes of hardware execution are available: simulation and silicon. In the simulation mode, ModelSim is launched on the same machine and Inspect communicates dynamically with ModelSim through a TCP connection and `tcl` scripts. To "execute" a line of `C` source, Inspect directs ModelSim to simulate a specific number of clock cycles – the hardware must be simulated to the point that all computations on the line of source have been executed. To achieve this, Inspect uses the debug database described in the previous section. For each line of `C`, the database contains the corresponding LLVM IR instructions, and for each such instruction, the database contains the FSM state (clock cycle) in which it was scheduled by HLS. Consequently, with the debug database, we know precisely the clock cycles (FSM states) to execute for a given line of source. Naturally, as with software debugging, correspondence between the sequential `C` code and the hardware execution is strengthened when the HLS is run with compiler optimizations turned off. This may cause overhead in execution. However, if optimizations are on, as the HW executes cycle by cycle, the lines of `C` source being executed may appear to "jump around", based on the extent to which LLVM re-ordered the computations during optimization.

In silicon mode, Inspect uses Altera's SignalTap II [6] logic analyzer to observe signal values during hardware execution. The user must select a set of signals (discussed below), and Inspect then automatically generates a configuration file for

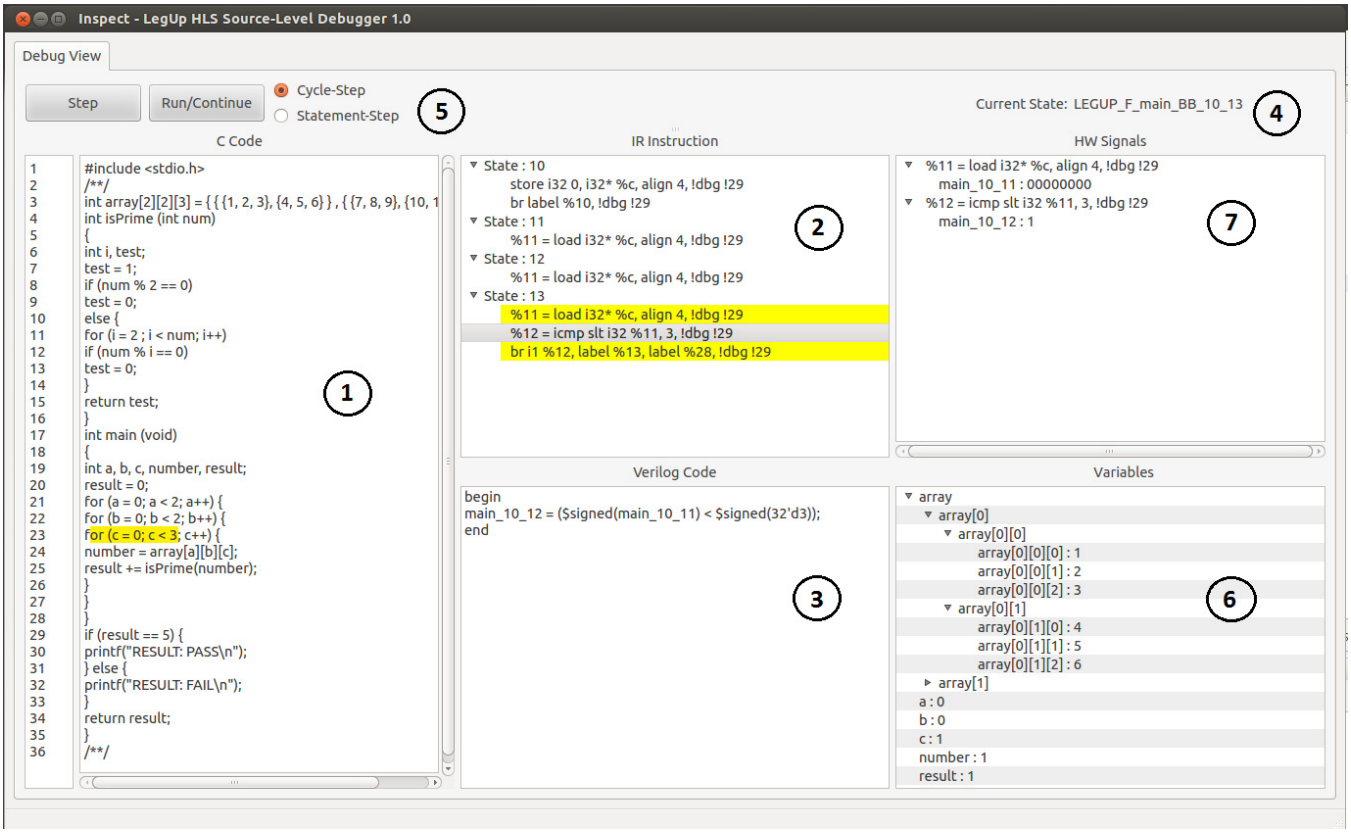

Fig. 5. Screenshot of Inspect during a debugging session.

Signal Tap, which specifies the list of signals to be observed during on-chip debug and the trigger criteria defining the conditions under which the signal values are captured. The HLS-generated circuit must be compiled using the Altera Quartus II Signal Tap flow, which instruments the circuit so the selected signal values are stored in on-FPGA block RAMs and transferred to the connected host PC when trigger criteria are met. Thus, the number of signals and the number of cycles for which they may be tracked is limited by the available on-FPGA RAM memory. To assist with these limitations, Inspect offers two facilities:

1) Sliding window-based debug: There are two parameters that define Inspect's trigger criteria: *start point* and *window size*. Inspect accepts a window size parameter from the user (# of cycles), defining a debugging window in which signals are observed during on-chip execution. The debug window moves forward iteratively, allowing Inspect to observe all required values for the selected signals. That is, the *start point* is automatically bumped up by the *window size* to gather the execution data from the next debugging window. The process of window-based debugging is automated and is performed by Inspect in the background.

2) Intelligent signal selection: There are normally *many* more logic signals in the hardware than there are variables in the software, and it is not necessary to expose all hardware signals to the user. For example, consider a statement such as `z = a + b;`, where, in the hardware, the value of `z` is registered and as such, for the signal variable `z`, there must exist a set of signals on the inputs to the register and a set of signals on the output of the register. Using the debug database, Inspect shows the user a list of the signals whose values may be changed in any given clock cycle, thereby assisting the user in narrowing down the signals to capture in on-chip debug. Note that the signals specifying the FSM state, as well as those attached to the LegUp system's memory controller are always monitored in silicon mode.

Given the above, it is worth underscoring a key difference in the visibility of data between the simulation and silicon modes: In simulation mode, the user can inspect the value of *any* variable in the HLS-generated circuit. This is possible as the RTL is being executed in ModelSim and Inspect can query the simulator to return the value of *any* logic signal in *any* particular simulation cycle. Conversely, in the silicon mode, the user must pre-select those signals of interest, subject to the constraints on limited on-chip block RAM. Only the values of pre-selected signals can be viewed as the user debugs with Inspect. If the user wishes to change the set of signals to be monitored, a new Signal Tap configuration file must be generated, and the design must be resynthesized by the Quartus II Signal Tap flow.

To underscore the need for silicon debugging and to explain the reason why functional RTL simulation is used instead of timing simulation, Table I shows the run-time needed for functional RTL simulation (non-timing-driven) in comparison to a post-routing timing simulation for 4 CHStone benchmarks [11]. Column 2 shows the run-time, in seconds, for RTL simulation with ModelSim-SE; column 3 shows the run-time for timing simulation; column 4 shows the increase in run-time

TABLE I. SIMULATION RUN-TIME COMPARISON.

Benchmark	RTL Sim (s)	Timing Sim (s)	Increase (×)
MIPS	0.9	76	84
MOTION	1	118	118
DFMULL	1.6	248	155
DFADD	0.8	37	46

between RTL and timing simulation. As illustrated, timing simulation takes roughly 2 orders of magnitude longer than RTL simulation, and as such, is plainly impractical for large designs with lengthy vector sets. Moreover, there are undoubtedly bugs which cannot be discovered even with a timing simulation, for example timing issues arising from process variations, transient functional bugs from SEUs, and others. Hence, we feel there is a strong need for HLS debugging strategies that leverage actual silicon execution, such as available with Inspect.

B. Execution Management

When debugging commences, Inspect executes the hardware for a single clock cycle and extracts the FSM state (accessible in both simulation and silicon modes). The current FSM state is the crucial timestamp used by Inspect to determine precisely "where" the HLS-generated HW is during execution – analogous to the program counter in a microprocessor — displayed at label ④ in Fig. 5. Given the FSM state, the debug database permits access to the corresponding variables, signals, LLVM IR, `C` statements and functions. Inspect offers stepping and breakpointing capabilities that are similar to those found in software debuggers.

1) Stepping: Two flavors of stepped execution are available to the user, which we call *statement step* and *cycle step*. Statement step mimics software-like stepping, where the execution is advanced one `C` statement at a time. With each statement step, the hardware execution is advanced by one or more clock cycles until the statement is completed. Cycle step, on the other hand, advances the hardware execution by a *single* clock cycle, which we expect will be useful to hardware engineers to gain insight into the HLS-generated HW. With cycle step, for example, a user can observe the cycle-by-cycle execution of a single `C` statement. The two different styles of stepping can be used at any time, based on the user's needs, through the options shown at label ⑤ in Fig. 5. LegUp generates a Verilog module for each `C` function, and instantiates the module in the parent (calling) function's module. Inspect supports both stepping *into* or *over* function "calls" (invocations of Verilog sub-modules).

2) Breakpoints: Breakpoints are widely used by software developers to control program execution and spot bugs close to a specific location. With Inspect, users can enable/disable one (or more) breakpoints at the `C` source level. The familiar run/continue functionality of software debuggers is available to execute the hardware until reaching a breakpoint (see buttons left of label ⑤ in Fig. 5), and then continuing to the next breakpoint. Breakpoints are set by clicking on the line number on the left of panel ① in Fig. 5.

C. Inspecting Signals

Users can observe both software variables and hardware signals in Inspect, displayed at locations ⑥ and ⑦ in Fig. 5, respectively. Hardware signals are presented in hexadecimal form, and are likely most useful to hardware engineers seeking low-level insight into the hardware behavior. Conversely, the values of software variables, while extracted from the HW execution, are presented to the user in the typical software form. These are useful to those wishing to debug at the highest abstraction level. Both software variable and hardware signal lists are filtered based on the active scope of the program being debugged. For example, in case of multi-function programs, users only see the variables and signals that are related to the current executing function. Global variables are also detected by Inspect and are shown during the entire design execution. Note that LegUp does not support `malloc` and `free` and hence, there is no heap-allocated data to display.

Data types for software variables are detected automatically by Inspect. Users can observe complex structures that include *arrays* or composite types such as *structs*. Inspect also converts all the variables to their actual data type and presents them according to their type. For example, *float* and *double* fields are shown as floating point values. Observe at window pane ⑥ in Fig. 5 the current execution state of the program/circuit, including the values of the 3D global variable `array`.

D. `gdb` Integration for HW/SW Discrepancy Detection

Besides the simulation-based and silicon debug ability, Inspect is also capable of debugging purely in software mode through integration with the `gdb` debugger [15]. Inspect uses `libmigdb` library [13] to connect to `gdb` and controls the debug session. From this perspective, Inspect can be seen as a GUI on top of `gdb` debugger, allowing the user to issue general commands such as `run`, `step`, etc. The main motivation, however, for the `gdb` integration is to permit HW/SW discrepancy detection.

With HLS, as the user is starting with a high-level program specification, he/she will have already tested the software prior to HW synthesis. Given buggy hardware, we believe it would be valuable to the user to rapidly identify the first point at which the hardware execution result differs from the software – the first discrepancy. To realize this functionality, Inspect concurrently executes the hardware *and* the software (using the integrated `gdb`), comparing the two as they execute. Variable values in `gdb` are compared with their corresponding HW signals, and inconsistencies are identified. The debug database allows us to sync between the HW and `gdb` executions when each are stepped, one in a statement-by-statement manner; one in a cycle-by-cycle manner. In essence, we use `gdb` to step a `C` statement, and then (using the debug database) we step the HW for a number of cycles until the computations of the `C` statement are completely executed in HW, facilitating a direct comparison. The user is presented with a list of "bugs" (discrepancies) and pointed to the precise line of `C` source where the issue occurred.

E. RTL/Silicon Discrepancy Detection

In addition to the ability to compare the SW to RTL, we include the ability to compare the RTL to the silicon. In this flow, ModelSim and the Signal Tap-instrumented hardware are run in tandem, and the first mismatch between the two are identified. Since the FSM state is always tracked in silicon

mode, Inspect is able to show the user the corresponding C code at the root of the mismatch.

We recognize that in Inspect's current form, it may be difficult for a software engineer to correct a bug detected by one of the discrepancy detection flows – such correction may require HW skills. The software engineer may, for example, consider re-writing the C in a different style in the hope that the resulting HLS-generated HW would be changed. A direction for future work is "Wizard-like" functionality within the debugger to suggest code fixes to the user for bugs detected.

V. Experiments

The previous section described the features of Inspect via a case study in a manner that demonstrated the usefulness of the platform. As with any research on debugging, it is difficult to numerically quantify the productivity improvements arising from new methodologies, such as those enabled by Inspect. In this section, we present an experimental study of the automated bug discrepancy detection, both SW vs. RTL, and RTL vs. silicon.

As a first experiment, we took 6 of the CHStone benchmarks [11] and synthesized them with LegUp 3.0, producing 6 functionally correct Verilog implementations. To represent a scenario in which the HLS tool produced incorrect Verilog, we created three "buggy" variants of each circuit, where we manually inserted bugs into the Verilog, by changing a single line. As a representative example, for the dfadd benchmark, the three "buggy" variants are as follows:

1) A state transition in the FSM was changed.
2) A value (returned by a Verilog module) that was originally AND'ed with 0x7FF was altered to be AND'ed with 0xFAA.
3) A value that was originally decremented was altered to be decreased by 10.

Similar changes were applied to the other 5 benchmarks. Note that the CHStone circuits have built-in input vectors and incorporate correctness checking of the computed results vs. golden outputs. Thus, we can "execute" the circuits and their correctness is assessed as part of the circuit itself.

Having generated 18 buggy circuits, we invoked Inspect's automated HW/SW discrepancy detection described in the previous section. The results are shown in Table II. Columns 1 and 2 show the name of each benchmark and the index of the buggy variant. Column 3 shows the number of bugs found by Inspect. Note that although each variant contains only a *single* bug, Column 3 shows the number of values computed within the circuit that *differ* from the concurrent gdb execution of the corresponding software program. Such differences are found *automatically* by our tool, and for each difference, Inspect uses the debug database to pinpoint precisely the line of C code where the difference arose.

Columns 4, 5 and 6 of Table II are perhaps the most important in illustrating the utility of this feature. Column 4 shows the number of HW cycles executed until the first bug was detected. Column 5 shows the total number of HW cycles for the HW execution to run to completion. Observe that, in the geomean row at the bottom of the table, $\sim 10\times$ fewer cycles (on average) are executed to detect the first bug vs. completing

TABLE II. Automated hardware/software discrepancy detection results.

Benchmark	Bug	# Bugs	1st Bug Cyc	Tot Cyc	Pass/Fail
MIPS	Bug #1	7579	503	16919	Fail
	Bug #2	16337	449	17582	Fail
	Bug #3	5497	1474	17684	Pass
GSM	Bug #1	13264	1785	14030	Fail
	Bug #2	16	27664	30424	Fail
	Bug #3	2817	6362	30642	Fail
MOTION	Bug #1	66	62	20212	Fail
	Bug #2	20	19425	19829	Fail
	Bug #3	88	18953	19816	Fail
DFMUL	Bug #1	25	138	3294	Fail
	Bug #2	3	479	3318	Fail
	Bug #3	21	588	3308	Fail
DFADD	Bug #1	51	168	8161	Fail
	Bug #2	735	68	10133	Fail
	Bug #3	2	400	8257	Pass
DFDIV	Bug #1	24	1851	6793	Fail
	Bug #2	112	1594	6769	Fail
	Bug #3	24	1808	6793	Fail
	GEOMEAN:	138.7	1046.4	10831.9	

the execution. For each bug, the user is shown the precise line of C code where the mismatch occurred. We expect that, without Inspect, a traditional debugging cycle would operate as follows: First, the user would execute the design to completion and check for an error in the overall circuit outputs. Finding such an error, the user would begin backtracking to narrow down the source of the error, likely by adding $display statements into his/her Verilog code. Multiple tedious manual modifications of the HLS-generated Verilog would be needed, along with associated HW executions until the precise location of the bug was found. Column 6 shows, that for 16 of 18 cases, the bug did indeed propagate to overall circuit outputs – such cases failed correctness checking. However, for 2 of the cases, the bug caused incorrect intermediate values, but those values did not propagate to overall outputs, likely because the built-in vectors for CHStone did not exercise certain portions of the circuit. Thus, the user would most likely presume (wrongly) that such circuits were bug free. Such bugs were nevertheless uncovered with Inspect, which we believe is a significant advantage of our approach vs. manual methods. In a second experiment, we evaluated the silicon debug mode. We introduced bugs into the LegUp-generated FSM of 4 benchmarks, creating three buggy version of each benchmark. Each bug consisted of changing one or more transitions in the FSM state diagram. For example, for the mips benchmark, the bugs are as follows:

1) Changed the destination state of a transition.
2) Reversed the state destinations corresponding to if/else.
3) Skipped over a state that impacted a value computed by a Verilog module.

We then used Inspect's silicon mode to find the bug in the FSM. This experiment reflects the scenario where Quartus II introduced bugs during synthesis/place/route, or where the bugs are within the silicon itself. We specify the current FSM state (in all FSMs) as being the HW signals to observe in Signal Tap. For this experiment, we setup Inspect to trace in windows of 2000 cycles, meaning that, we configure the Cyclone IV device, capture the FSM state for 2000 cycles

TABLE III. SILICON DEBUG RESULTS.

Benchmark	Bug	1st Bug Cyc (# Configs)	Total Cyc (# Configs)	Pass/Fail
MIPS	Bug #1	412 (1)	6552 (4)	Pass
	Bug #2	677 (1)	827 (1)	Fail
	Bug #3	6594 (4)	6640 (4)	Fail
MOTION	Bug #1	8355 (5)	9347 (5)	Fail
	Bug #2	208 (1)	566 (1)	Fail
	Bug #3	8357 (5)	9204 (5)	Fail
DFMUL	Bug #1	486 (1)	1766 (2)	Fail
	Bug #2	208 (1)	∞ (∞)	Fail
	Bug #3	768 (1)	1948 (2)	Pass
DFADD	Bug #1	234 (1)	∞ (∞)	Fail
	Bug #2	208 (1)	4103 (3)	Fail
	Bug #3	983 (1)	4117 (3)	Fail

(via Signal Tap), adjust the Signal Tap trigger condition, reconfigure the Cyclone IV device, capture the FSM state for the *next* 2000 cycles, and so on. For each window, Inspect automatically compares the sequence of FSM states extracted from the silicon, with those extracted from the circuit's RTL simulation in ModelSim. When a discrepancy is found, Inspect uses the debug database to point the user to the line of `C` code related to the faulty state.

The results for the silicon debugging experiment are shown in Table III. Column 3 shows the cycle number at which the first bug was detected; column 4 shows the total number of cycles for HW execution to complete. Column 5 indicates whether the design passed or failed the built-in correctness checking. The time required for debugging is closely tied with the number of device reconfigurations required – this is shown in parentheses in columns 3 and 4. Observe that for 7/12 cases, Inspect found the first bug in fewer device configurations than required to run the design to completion. For two cases, the buggy design actually passed correctness checking, though in both of such cases, Inspect found the bug in the first trace window. For two different cases, the total number of HW cycles is shown as ∞, as in these cases, the FSM bug caused the HW to run indefinitely. In both such cases, Inspect found the bug in the first trace session. We believe engineers will find it quite useful to rapidly identify HW bugs that: 1) do not propagate to circuit outputs, and/or that 2) cause the HW not to terminate.

VI. CONCLUSIONS AND FUTURE WORK

A long-term vision for high-level synthesis is to make the speed and energy benefits of hardware accessible to software engineers, and significant progress has been made in recent years in the quality of circuits produced by HLS. What is needed to realize that vision, however, are debugging tools that offer a software engineer's perspective into the auto-generated hardware. The Inspect framework proposed in this paper provides `C` source-level debugging for HLS-generated hardware, giving the user familiar step, run and break execution management, and the ability to inspect variables, where the values are drawn from the hardware: either RTL simulation or execution in silicon. The Inspect GUI relates three levels of program/circuit abstraction: `C`, LLVM IR, and Verilog. Beyond traditional `gdb`-like functionality, Inspect provides automated discrepancy detection between HW and SW for quick pinpointing of HW/SW differences. Inspect will be incorporated into the next release of the open-source LegUp framework.

Future work involves adding support for watchpoints, improving the correlation between software and hardware when the hardware is generated with compiler optimizations turned on, and debugging support for hybrid systems that contain both a processor and HLS-generated accelerators. Supporting LegUp loop-pipelining feature will be also considered as future work.

ACKNOWLEDGMENT

The authors gratefully acknowledge the financial support of Altera Corporation and the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors also thank the entire LegUp research group for many helpful technical discussions.

REFERENCES

[1] C-based design: High-level synthesis with the vivado hls tool. Technical report, Xilinx Incorporated, 2013.

[2] Catapult: Product family overview. Technical report, Calypto, 2014.

[3] Certus ASIC prototyping debug solution. Technical report, Mentor Graphics Corp., 2014.

[4] Chipscope pro and the serial i/o toolkit. Technical report, Xilinx Incorporated, 2014.

[5] Altera, Corp., San Jose, CA. *Cyclone-IV Data Sheet*, 2010.

[6] Altera, Corp., San Jose, CA. *SignalTap II Logic Analyzer User Guide*, 2013.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, , and J. H. Anderson. LegUp: An open source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Tranasactions on Embedded Computing Systems*, 2013.

[8] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *ACM DAC*, pages 433–438, 2006.

[9] P. Coussy, G. Lhairech-Lebreton, D. Heller, and E. Martin. GAUT a free and open source high-level synthesis tool. In *IEEE DATE*, 2010.

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

[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[12] K. S. Hemmert, J. L. Tripp, B. L. Hutchings, and P. A. Jackson. Source level debugger for the Sea Cucumber synthesizing compiler. *IEEE FCCM*, pages 228–237, 2003.

[13] http://libmigdb.sourceforge.net/. *GDB/Machine Interface library*, 2014.

[14] http://www.altera.com/products/software/ opencl/opencl-index.html. *OpenCL for Altera FPGAs*, 2013.

[15] http://www.gnu.org/software/gdb/. *GDB The GNU Project Debugger*, 2014.

[16] http://www.llvm.org. *LLVM Compiler Infrastructure Project*, 2014.

[17] http://www.mysql.com. *MySQL Open Source Database*, 2014.

[18] http://www.xilinx.com/products/design_ tools/vivado/vivado-webpack.htm. *Xilinx: Vivado Design Suite*, 2013.

[19] J. L. Tripp, P. A. Jackson, and B. L. Hutchings. Sea Cucumber: a synthesizing compiler for FPGAs. *FPL*, pages 875–885, 2002.

[20] K. Wakabayashi and B. Schafer. All-in-C behavioral synthesis and verification with CyberWorkBench. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 113–127. 2008.

[21] H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow. High-level synthesis with behavioral level multi-cycle path analysis. In *IEEE FPL*, 2013.