# An FPGA Overlay Architecture Supporting Rapid Implementation of Functional Changes during On-Chip Debug

Al-Shahna Jamal
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
alshahnaj@ece.ubc.ca

Jeffrey Goeders
Electrical and Computer Engineering
Brigham Young University
Provo, Utah, USA
jgoeders@byu.edu

Steven J.E. Wilton
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
stevew@ece.ubc.ca

***Abstract*—As Field-Programmable Gate Arrays become more complex, debugging designs implemented on these devices has become increasingly time-consuming. For many types of bugs, simulation is not sufficient, and the only way to uncover the root cause of unexpected behaviour is to run the design in hardware at speed. Many techniques that support on-chip debug have been described; typically, these techniques involve instrumenting the design to increase observability. In this paper, we describe instrumentation that not only increases observability, but that can also be used to control certain aspects of the design. Supported functional changes include applying small deviations in the control flow of the circuit, or the ability to override signal assignments to perform efficient "what if" tests. Our approach uses a novel overlay architecture which allows these changes to be implemented during debug without recompiling the design. Changes can be made in seconds, dramatically reducing the time to perform a debug iteration. Our overlay is specifically optimized for designs created using a high-level synthesis (HLS) flow; by taking advantage of information from the HLS tool, the overhead of the overlay can be kept low.**

## I. Introduction

As the capacities of Field-Programmable Gate Arrays (FPGAs) grow, the complexity of systems implemented on FPGAs is increasing. For complex designs, debugging is difficult and time-consuming. When unexpected behaviour is observed, finding the root cause of this behaviour is challenging. Simulation is an essential tool for finding the root cause of many bugs, however, there are many situations in which bugs do not manifest themselves until the design is run in hardware. This may be due to interactions with the environment that can not be effectively modeled in simulation, interfaces with legacy IP blocks for which simulation models are not available, or due to long run-times which may prevent bugs from manifesting in any reasonable simulation run. To find the root cause of these sorts of bugs, the only solution is to execute and observe the design running in hardware, at speed.

Debugging a design running in hardware is complicated by the fact that I/O pins are limited. Previous work has proposed instrumenting a design with *trace buffers* which are on-chip memories that store the behaviour of selected signals as the circuit executes. Captured data from the trace buffers can then be used to replay the behaviour offline. Commercial implementations of this idea are available [1], [2]. Recent academic work has extended this by focusing on designs developed using high-level synthesis (HLS) flows [3]–[5]. In particular, [5] leverages information from the HLS tool to significantly compress how data is packed into the trace buffers, making effective use of limited on-chip memory.

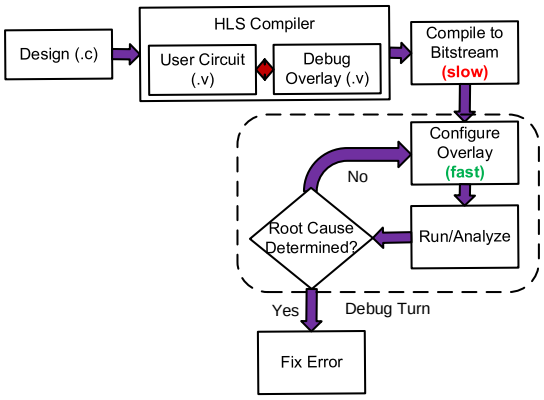



Fig. 1: Overall Approach

However, one main challenge with in-system debug is the recompilation that is required any time the debug instrumentation is changed. For example, in [5], the instrumentation needs to be reconstructed each time the set of variables to be recorded changes, requiring a full recompilation (including place and route). A full recompilation is time-consuming, and limits debug productivity. In [6], it is shown that software-like debug turn-around times (on the order of hundreds of milliseconds) can be achieved by instrumenting the design with a flexible *overlay*. Between debug turns, the overlay can be reconfigured, allowing the instrumentation to record a different set of variables, without requiring a recompilation of either the instrumentation or the user circuit.

The overlay in [6] is designed to passively monitor the circuit and record behaviour. This is valuable to help the user

understand the behaviour of a design. However, there are many situations in which the user may wish to *control* certain aspects of the circuit as it runs. Examples of these functional changes may be small deviations in control flow to avoid problem areas during debug, or the ability to override signal assignments to perform efficient "what if" tests. In this paper, we show how an overlay similar to that in [6] can be extended to provide this capability. Specifically, the contributions of this paper are:

1) We describe a flow in which a user can not only observe the behaviour of important variables while running an HLS-generated design in hardware as in [6], but can also make small changes to the functionality of the design to aid debugging. Importantly, these changes can be made *without requiring a recompilation of the design or instrumentation*.
2) The flexibility to make these changes comes from instrumentation in the form of a flexible overlay that is added to the design at compile time. We describe several alternative architectures for such an overlay that balance flexibility and overhead.
3) We present experimental results that quantify to what extent the user can change the behaviour of a design, as well as the area and delay overhead incurred when using the overlay.

This paper is organized as follows. Section II discusses related work. The baseline flow upon which we build our work is described in Section III. Section IV discusses the capabilities supported by our overlay and Section V presents the architecture. Results are presented in Section VI and challenges related to circuit optimizations are discussed in Section VII. Section VIII concludes the paper.

## II. Related Work

Earlier work that supported applying functional changes on-chip can be found in [7]–[9]. Techniques from [7], [8] employed rectification algorithms to change the functionality of an existing LUT in the design to have the new desired behaviour, and required a recompile if the functional change did not fit within the existing netlist connections. Our work has similarity to [9] which instruments registers in the design with multiplexers in order to override signals at runtime. However, [9], [10] use formal verification techniques to perform automatic bug detection; this is orthogonal to our work which instead leverages the debug instrumentation to provide information about the design to the user, to help him or her root cause observed errors.

Our debug instrumentation takes the form of an overlay to allow rapid configuration of the debug logic at runtime. Similar solutions are proposed in [11], [12], however these works target generic RTL designs and focus on increased signal observability. Like [3], [4], [13] our work provides source-level debugging for HLS designs such that the user need only interact with their original source code. However, our debug overlay is also capable of applying functional changes to the underlying circuit that specifically map back to actual constructs in the source code, providing the designer with an increased understanding of how their software affects the generated hardware.

## III. Baseline Debug Flow

To put our work in context, this section describes the baseline debug flow upon which we build our work. The flow is similar to that described in [6].

The baseline flow is shown in Figure 1. The HLS tool (we use LegUp [14]) is augmented with the ability to create debug instrumentation and add this instrumentation to the user circuit. The instrumentation is optimized for the specific user circuit being compiled; in [5], it is shown that by taking advantage of the scheduling information for the circuit, the trace data to be stored can be significantly compressed (127x in [5]). The instrumentation is added to the RTL description of the circuit; a *debug database* generated by the HLS tool is used to provide a mapping from user-visible C variables and RTL registers. Back-end vendor place and route tools (we use Quartus Prime) are then used to compile the user circuit and instrumentation, producing a bitstream.

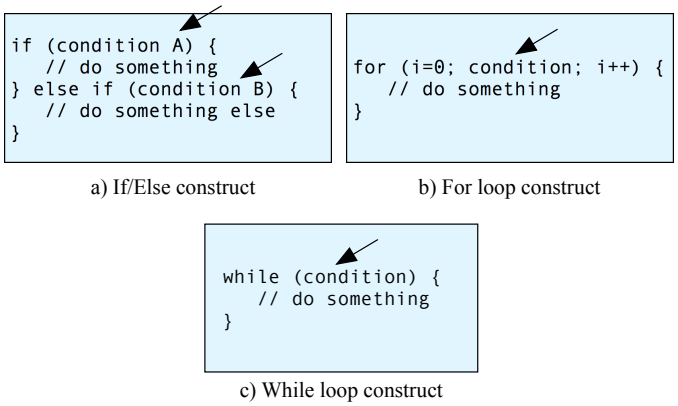
At run-time, the FPGA runs as normal, at speed. The instrumentation records the behaviour of selected signals into on-chip memory (trace buffers) configured as circular buffers. After the run has completed or when a breakpoint is encountered, the data in the trace buffer can be read out using the JTAG port. Unlike commercial RTL tools such as Chipscope or SignalTap II which present data using waveforms [1], [2], we allow the user to replay the execution using a software-like debug interface, similar to [5]. The user can single step forwards and backwards through the original source code, observing the values of variables at each step. The user can use this information to understand the behaviour of the design as it was executing at speed.

Often, multiple runs (called *debug turns*) will be required to uncover the root cause of observed unexpected behaviour. Between each turn, the user may wish to change the set of variables that are recorded, or change the breakpoint used to halt the design. Collectively, we refer to the set of user variables to be recorded and the breakpoint location and condition as a *debug scenario*. The user configures the instrumentation to implement the debug scenario and re-runs the design. Importantly, a full compilation is not required. The instrumentation is flexible, containing memory bits that can be adjusted to implement the desired debug scenario. As in [6], these bits can be configured in seconds, providing software-like debug turn-around times. Because of the flexible nature of this instrumentation, we refer to it as an *overlay*.

The user can perform debug turns as many times as desired, each time possibly using a different debug scenario as his or her understanding of the design evolves.

## IV. Debug Capabilities

The instrumentation overlay described as part of the baseline flow in Section III is limited to passive observation of the user circuit. In contrast, the overlay described in this paper provides the ability to make small changes to the functionality of the




a) If/Else construct b) For loop construct

c) While loop construct

Fig. 2: Supported Control Flow Changes. Arrows represent conditional branches that can be modified.

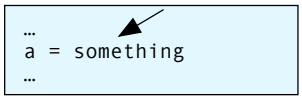



Fig. 3: Variable assignment override. The value assigned to a variable can be modified.

user circuit to further accelerate debugging. In this section, we first describe three example use cases that motivate our approach, and then provide a more precise description of the capabilities of our overlay.

### A. Example Use Cases

The following describes three use cases that motivate our approach:

**Use Case 1**: A circuit is crashing because an alarm signal goes high erroneously. The user understands the reason, and will fix it later, but wants to let the circuit 'limp along' to continue debugging.

**Use Case 2**: The user has determined that a particular loop is problematic. The user wants to skip the loop entirely, since it is not germane to what he or she is worried about right now.

**Use Case 3**: While debugging, the user is puzzled by a particular assignment to a variable in a segment of code written by someone else. He or she wants to do a quick "what if" test by changing the value assigned to the variable, in an attempt to better understand the code.

In all three of these use cases, the functionality of the user design needs to be modified before the next debug turn. Although it would be possible to edit the source code and recompile, our approach allows the user to make these small changes by reconfiguring the overlay.

### B. Overlay Capabilities

Our overlay is flexible enough to implement both passive monitoring of the user circuit as well as to implement changes to the functionality of the design to support the types of use cases in Subsection IV-A. The passive observation capabilities consist of the ability to allow the user to configure (1) the set of user variables to record in the trace buffer, (2) functions for which user variables and control flow information should be recorded, and (3) a conditional breakpoint where the condition itself is programmable. These capabilities are very similar to those described in [6], and thus are not described further here. In the rest of this section, we focus on the capabilities related to implementing small functional changes that we have chosen to support in our overlay.

Our overlay supports two types of changes to the functionality of the user circuit: changes to the control flow and changes to individual variable assignments. Each is described below.

*1) Control Flow Changes:* Use Cases 1 and 2 in Subsection IV-A require the ability for the user to change the control flow of the design between debug turns. For reasons that will become clear in Section V, rather than providing the user the ability to arbitrarily modify control flow between any two lines of C code, our overlay restricts the control flow transitions that can be modified to those that occur as the result of a conditional branch that stems from 'if/else', 'for', or 'while' constructs in the user's C code. More specifically, as shown in Figure 2, for an 'if' construct, the condition related to the 'if' clause as well as any conditions related to 'else if' clauses can be modified. For a 'for' or 'while' loop construct, the condition that ends the loop can be modified.

We consider two variants of this control flow capability. In the first variant, the user can statically indicate the outcome of a branch to either always be taken, or never be taken. In the second, the overlay is architected such that the user can impose a condition that must be satisfied before altered control flow is imposed onto the user circuit. This means that until this condition is true, the user circuit will operate as normal. An example use case of this capability would be to exit a loop early (it may be sufficient to trace only a few iterations of the loop in order to understand the loop's behaviour). We limit the "support" variables for the user condition (that is, the variables upon which the condition is based) to be the same as, or a subset of, the variables in the original condition, For example, if a "for" loop was written to exit when variable $i$ reaches a certain value, only $i$ can be used in the replacement condition. This design decision will be justified in Section V.

HLS tools typically optimize the code before generating hardware. In some cases, loops written in the C code may be translated into predicated operations and then optimized in a way that there is no explicit condition variable. In such cases, the user would not be able to alter the control flow for such constructs. We anticipate a GUI would indicate which constructs can be instrumented and which can not; in Section VI, we will quantify what proportion of constructs can be instrumented in our benchmark circuits.

*2) Variable Assignment Override:* Use Case 3 in Subsection IV-A requires the ability to change the value of an assignment (Figure 3). As will be shown in Section V, providing this capability is more intrusive on the user circuit than providing the capability to change the control flow. Thus, we require the user, *at compile time*, to select which variables he or she would

like to override, and our tool instruments all the assignments to these variables (providing the user with the flexibility to alter the variable at any point in the program). At run-time, the user can override any of the assignments that have been instrumented. For efficiency reasons, the *replacement* value for the right-hand side of the assignment must be a constant. If the user wishes to override an assignment that has not been instrumented, he or she needs to modify the source code and recompile. While this falls short of the debug experience that software designers might like (being able to override *any* assignment in the code), it provides a balance between area/delay overhead and flexibility.

As in Subsection IV-B1, we consider two variants of the architecture. In the first variant, the circuit modification is static, meaning that if an assignment is overridden, every time the assignment occurs, the replacement value is used. In the second variant, the replacement value is only used if a specified condition is true; if the condition is false, the circuit operates as normal.

## V. Overlay Architecture

The key technical challenge in our approach is creating an architecture for the overlay that is flexible enough to support the types of changes described in Section IV, yet has small area overhead and has as little impact on the clock frequency of the user circuit as possible.

Recall from Section III that the overlay is constructed by the HLS tool when the C code is compiled. Thus, similar to [6], we can use scheduling and variable information within the HLS tool to construct the overlay on a circuit-by-circuit basis. In the following subsections, we describe our overlay architecture that supports both control-flow modifications and variable assignment override, and show how the overlay can be adapted to the number of control flow constructs and selected variable assignments in the source code. Throughout, we assume that these capabilities are being added to an overlay that supports passive observation, such as the overlay in [6].

### A. Architecture for Control Flow Changes

**Control Flow (CF) Variant - No Conditional Support**

As discussed in Section IV, support for this capability involves being able to *configure the outcome of a conditional branch operation*. When compiling the input C code, most HLS tools first translate the source code to an internal representation. In our case, we use Legup, which translates C code into LLVM's Intermediate Representation (IR). Backend routines are then used to translate the IR to hardware. By identifying the conditional temporary register in the IR, it is possible to determine the hardware signal that corresponds to this conditional temporary register. Our HLS tool identifies all such conditional temporary registers, and for each, inserts a multiplexer into the path of the corresponding hardware signal in the user circuit. This is shown in Figure 4. In this figure, each *original_br_sig* represents one of these signals.

The select lines of each multiplexer as well as the lower data input of each multiplexer comes from a memory that is

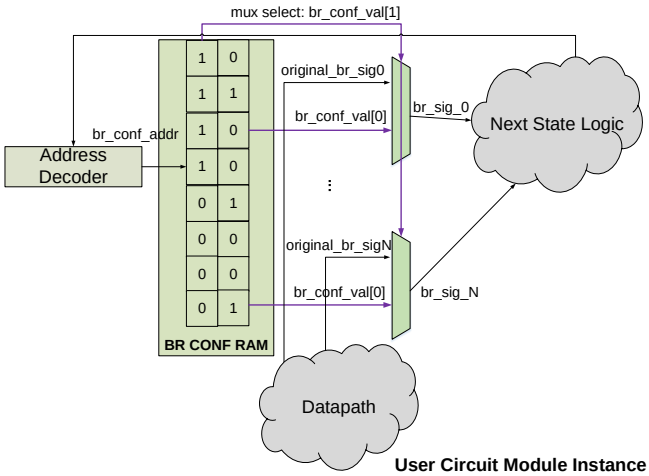



Fig. 4: Control Flow Instrumentation for Branches

part of the instrumentation. The memory contains one word for each conditional branch. Each word is two bits; the first bit is used to control whether the corresponding conditional branch should be overridden, and the second bit is the value to use if the branch is overridden. The memory is implemented in one or more on-chip memory blocks. The user can update the contents of this memory before a debug turn without requiring a recompilation using vendor tools (we use Intel's In-System Memory Content Editor). The address line of the memory, labeled *br_conf_addr* in Figure 4 is asserted in a state previous to the branch evaluation state in question to account for the 1-cycle latency of the memory.

Note that an alternative, more flexible, implementation is possible. Rather than providing the ability to add a multiplexer to each branch condition signal, it would be possible to create an overlay in which the multiplexer is added before each state bit, allowing the user to implement arbitrary state transitions without performing a recompilation. We chose not to implement such an architecture for two reasons. First, the area overhead would be significant; the on-chip memory would require one word per state, and each word would need to contain at least $log_2$ of the number of states. Second, HLS tools typically perform aggressive scheduling optimizations in which multiple lines of code are mapped to the same state. If a user created a transition in an attempt, for example, to skip an arbitrary line of code, it would be likely that additional lines of code would be skipped since they are mapped to the same state, creating confusion. By limiting the control changes to those constructs that have a clear relationship between the hardware and source code, we can avoid this complexity.

**Conditional Control Flow (CCF) Variant**

In the Conditional Control Flow variant (which we refer to as CCF), the user can specify a condition that, when true, allows the control flow to be modified. For clarity, the term *branch condition* refers to the IR variable and corresponding hardware signal that indicate whether a particular branch is taken, and the term *user condition* refers to the condition

the user specifies to determine whether a particular branch condition should be overridden during execution.

The overlay architecture for the CCF variant has two parts. The first part is a programmable comparison circuit that monitors variables in the user circuit. We use an enhanced version of the conditional buffer freeze (C) unit from [6] for this part of the overlay. The enhanced C unit, shown on the left-side of Figure 6(a), contains a number of programmable fields which the user can configure to indicate the variable to be used as part of the user condition, the state in which the variable is updated, and the type of comparison used in the user condition. As in [6], the supported operations are $=$, $>$, $<$, $\geq$, $\leq$, and $\neq$. The user can configure these fields through the debug UART port without requiring a recompilation.

Now, we need to restrict the variables that can be used in the *user condition* to the "support" variables used in the original *branch condition*. This is due to the aggressive scheduling operations performed by many HLS tools. As an example, if the *branch condition* we wish to override depends on variable $a$, then if we use a different variable $b$ in the *user condition*, it is possible that scheduling operations may update $b$ after the original branch is taken. In this case, the user condition will not affect how the branch is evaluated. By requiring both the *user condition* and *branch condition* to use the same set of variables, this can be avoided.

Many times these "support" variables or expressions in the *branch condition* do not map back to source code variables, and are instead intermediate or temporary values. Tracing these in addition to signals that map back to source code variables significantly increases the area of the trace scheduler. Therefore, we employ signal reconstruction from [5] to minimize the number of signals traced, with the constraint that all "support" variables must be part of the base set of traced signals.

The second part of the overlay is shown on the right side of Figure 6(a). This part is the same as in the original Control Flow (CF) variant, containing a memory that can be used to describe whether each *branch condition* can be overridden, in this case, under control of the *user condition*. The read enable signal of the memory is driven by the output of the C unit; if the read enable is zero (meaning the user condition is not satisfied) the memory output is zero. Since one of memory word bits drives the select lines of the multiplexers, in this case, the multiplexer selects the original branch condition from the user circuit, meaning the user circuits operates as normal. If the C unit output is one, the memory is read as before, possibly causing the branch condition to be overridden depending on the contents of the memory.

Note that the overhead of the C unit is large (in [6], the C unit adds approx. 200 ALMs to the instrumentation). However, in our implementation, we are adding our overlay to an overlay similar to the one in [6], meaning we can re-use the existing C unit for our purposes here. This requires adding a *mode* bit and a small amount of extra logic (shown in Figure 6(a)), and means that the user can not perform a conditional breakpoint and a conditional branch override at the same time.

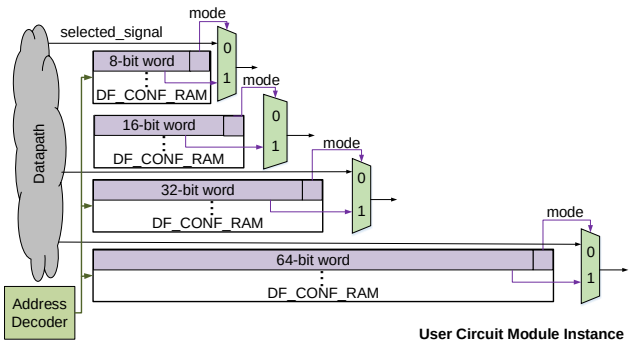



Fig. 5: Data Assignment Override (Dataflow) Overlay

### B. Editing Selected Variables at Runtime

**Dataflow (DF) Variant - No Conditional Support**

To use this capability, the user must first select variables they may be interested in overriding at compile time (we refer to this as the Dataflow (DF) overlay variant). Our tool determines all of the RTL signals that map to assignments to these variables and instruments only these in a similar fashion to the conditional branch instrumentation. Since data assignments can have different bit widths, our tool will size appropriate memories for 8-bit, 16-bit, 32-bit and 64-bit words per user circuit module as shown in Figure 5. Each word in the memory corresponds to one variable assignment. Within each word, one bit is used as the select line for the corresponding multiplexer, and the other bits are used to contain the override value. As with the conditional branch instrumentation, the contents of these memories can be written between debug turns using vendor tools without requiring a recompilation.

**Conditional Dataflow (CDF) Variant**

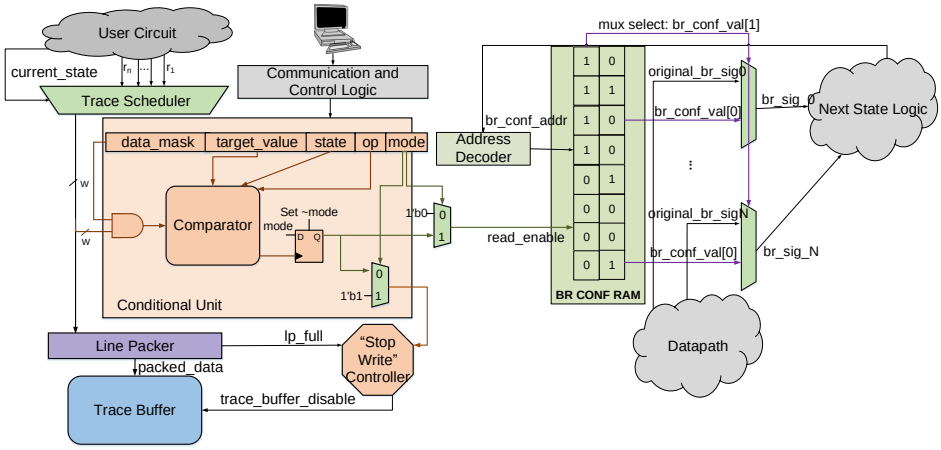
A conditional version of this architecture can be created in a manner similar to the Conditional Control Flow (CCF) variant in Subsection V-A. As shown in Figure 6(b), the C unit evaluates a condition, and the result is used to drive the read enable signals of the memories. In this way, if the condition is not true, no assignments are overridden, and the circuit operates as normal. We assume one C unit for all assignments; it would be possible to use multiple C units to selectively control different assignments, however, the overhead in doing so would be large.

## VI. Results

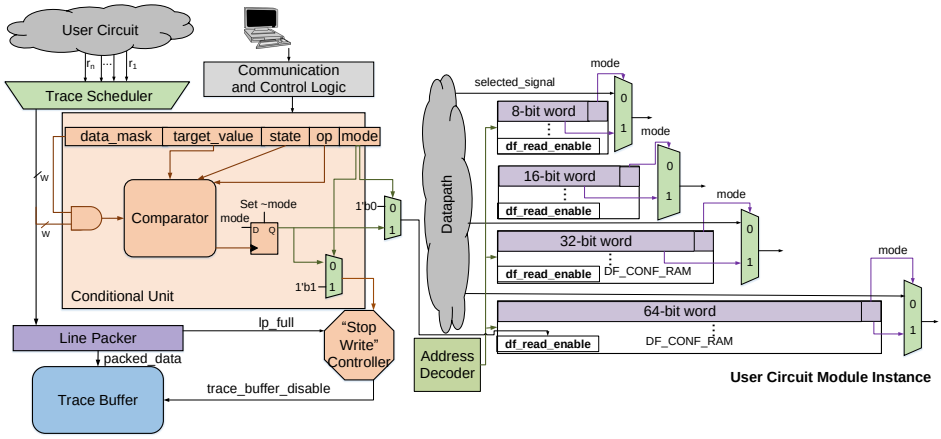
In this section, we evaluate the ability of the overlay to implement changes to the functionality of the user circuit, as well as the overlays' impact on area and clock speed.

### A. Control Flow Overlay Opportunities

In Table I, we quantify the number of control flow constructs in the hardware that the overlay is capable of altering at runtime to the total number of control flow constructs (for/while/if/else) available in the source code, for 13 circuits from the CHStone and MachSuite benchmark suites [15], [16]. These benchmarks were chosen in order to directly




(a) Conditional Control Flow (CCF) Overlay




(b) Conditional Dataflow (CDF) Overlay

Fig. 6: Conditional Overlay variants

compare with the previous work in [6]. In columns 6 to 8, the number of for loops, while loops and if/else statements are based on optimized (-O3) circuits. In order to provide a direct comparison of hardware support to the unique number software constructs, we count the instrumentation due to an inlined construct only once.

The number of hardware constructs in the right-most columns of Table I are lower than the software counterpart, due to compiler optimizations. The last column in Table I is the number of total overlay opportunities divided by total source code opportunities, 48.5% on average. The reason our overlay cannot alter 100% of these constructs is in cases of loops that have been completely unrolled, or if/else statements that have been promoted to ternary operations because in these cases, there is no branch signal available in the hardware to modify.

### B. Dataflow Overlay Opportunities

To quantify dataflow opportunities, we investigate the number of variables that our overlay is able to override. In Table II, column 2 shows the total number of variables for each benchmark (each inlined variable is counted separately because it is treated as a unique signal in the hardware and may also be optimized differently). Column 3 shows the number of variables that are optimized away; our overlay cannot alter these because there are no RTL sources to override. Columns 4 and 5 show the number of variables that are optimized to constants, and that reside in memories respectively. While we do not currently support overriding these types of variables, we could do so with more engineering effort. Finally, column 5 shows all the variables that reside in registers in the final hardware (on average 56.8% for our benchmarks) – these are the ones that can be altered by our dataflow overlays.

### C. Impact of Control Flow Overlays

Each of our benchmark circuits was compiled with LegUp which was modified to insert our overlay. The result was then mapped to a Stratix IV device using Intel's Quartus Prime. Table III shows the impact of CF and CCF on the maximum clock frequency (Fmax) as compared to the Base Overlay - these results are averaged over 3 fitter seeds. For Fmax, a 1ns clock constraint was specified for the benchmarks to encourage the tool to achieve a high-speed circuit implementation. The CF overlay only adds the branch instrumentation shown in Figure 4, while the CCF overlay integrates the branch instrumentation with the conditional unit shown in Figure 6b. From Table III, we can see that the CF overlay does not impact

TABLE I: Quantifying Overlay Control Flow Support for -O3 Compiled Benchmarks

| | Source Code | | | | Control Flow Overlay | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Benchmarks** | **Number of For Loops** | **Number of While Loops** | **Number of If/Else** | **Total Control Flow Opportunities** | **Number of For Loops** | **Number of While Loops** | **Number of If/Else** | **Total Overlay Opportunities** | **Overlay vs. Source Code** |
| adpcm | 17 | 0 | 29 | 46 | 4 | 0 | 3 | 7 | 15.2% |
| aes | 19 | 1 | 31 | 51 | 12 | 0 | 3 | 15 | 29.4% |
| blowfish | 5 | 5 | 13 | 23 | 3 | 4 | 2 | 9 | 39.1% |
| dfadd | 2 | 0 | 48 | 50 | 2 | 0 | 21 | 23 | 46.0% |
| dfdiv | 1 | 1 | 28 | 30 | 2 | 2 | 18 | 22 | 73.3% |
| dfmul | 2 | 0 | 26 | 28 | 2 | 0 | 15 | 17 | 60.7% |
| dfsin | 1 | 2 | 75 | 78 | 1 | 2 | 34 | 37 | 47.4% |
| gsm | 18 | 1 | 17 | 36 | 11 | 0 | 15 | 26 | 72.2% |
| jpeg | 34 | 11 | 74 | 119 | 19 | 7 | 49 | 75 | 63.0% |
| mips | 4 | 2 | 5 | 11 | 2 | 1 | 4 | 7 | 63.6% |
| motion | 7 | 5 | 27 | 39 | 1 | 2 | 9 | 12 | 30.8% |
| sha | 8 | 4 | 5 | 17 | 5 | 2 | 1 | 8 | 47.1% |
| FFT | 16 | 2 | 73 | 91 | 4 | 2 | 33 | 39 | 42.9% |
| **AVG** | | | | | | | | | **48.5%** |

TABLE II: Data Override Opportunities for -O3 Circuits

| **Benchmarks** | **Total Vars.** | **Opt. Away** | **Opt. to Const** | **Vars. in Mems** | **Vars. in Regs** | |
|---|---|---|---|---|---|---|
| adpcm | 249 | 72 | 16 | 32 | 129 | 51.81% |
| aes | 46 | 4 | 4 | 19 | 19 | 41.30% |
| blowfish | 42 | 1 | 10 | 16 | 15 | 35.71% |
| dfadd | 128 | 1 | 14 | 5 | 108 | 84.38% |
| dfdiv | 131 | 2 | 26 | 4 | 99 | 75.57% |
| dfmul | 84 | 2 | 19 | 4 | 59 | 70.24% |
| dfsin | 328 | 1 | 60 | 4 | 263 | 80.18% |
| gsm | 82 | 1 | 21 | 7 | 52 | 63.41% |
| jpeg | 263 | 22 | 14 | 64 | 163 | 61.98% |
| mips | 21 | 3 | 2 | 3 | 13 | 61.90% |
| motion | 100 | 11 | 54 | 8 | 27 | 27.00% |
| sha | 36 | 5 | 12 | 10 | 9 | 25.00% |
| FFT | 766 | 2 | 295 | 10 | 459 | 59.92% |
| **AVG** | | | | | | **56.80%** |

TABLE III: Impact of Control Flow Overlay Variants on Fmax

| **Fmax (MHz)** | **Base** | **CF** | **CCF** |
|---|---|---|---|
| adpcm | 123.60 | 127.27 | 127.09 |
| aes | 127.67 | 130.22 | 124.04 |
| blowfish | 167.63 | 175.26 | 160.57 |
| dfadd | 194.25 | 188.84 | 170.00 |
| dfdiv | 191.06 | 186.27 | 181.22 |
| dfmul | 182.95 | 180.19 | 177.53 |
| dfsin | 161.79 | 165.55 | 146.42 |
| gsm | 167.70 | 162.67 | 154.86 |
| jpeg | 96.39 | 90.76 | 87.10 |
| mips | 170.59 | 169.55 | 171.50 |
| motion | 153.14 | 151.45 | 151.51 |
| sha | 216.33 | 219.69 | 224.34 |
| FFT | 89.50 | 91.20 | 86.97 |
| **AVG +/- SD** | **152.32 +/- 5.04** | **152.02 +/- 4.38** | **146.13 +/- 4.78** |

average Fmax compared to the Base overlay (152.32MHz versus 152.02MHz), but there is variation across the circuits. This is because the branch RTL signals may lie on existing critical paths for only some of the benchmarks (eg. dfadd), and in these cases the branch instrumentation may cause increased delay on these paths. The CCF overlay drops the Fmax to 146.13MHz on average, and this is caused by the conditional unit which now drives the read_enable port on the configurable branch memories. In terms of fitter runtime, the average place and route for the original user circuit is 377.35 seconds for our benchmarks, and this increases to 525.47 seconds with the insertion of the CCF overlay.

In Figure 7, all area numbers shown are the *relative impact* when compared to the Base Overlay. The left most bars on the graph show the impact of the CF and CCF overlays on various components of the instrumentation. The branch instrumentation (labeled "functional override" on the graph) is on average 122 ALMs for the CF overlay, and 275 ALMs for the CCF overlay. The former is the logic for the branch instrumentation multiplexers, while the latter accounts for the increased number of "taps" from the user circuit to the existing trace scheduler. Recall, the branch "support' signals only need to be traced in the conditional (CCF) variant so that the Conditional unit has access to this data; we can see that the trace scheduler increases by 307 ALMs in this case. The impact on the content editor is due to the increased number of configuration bits in the overlay for this branch override capability compared to the Base Overlay.

### *D. Impact of Dataflow Overlays*

The area overhead of the dataflow overlay depends on how many variables the user has selected for instrumentation. In Figure 7, the dataflow overlays DF and CDF are constructed assuming 10% of variables are selected. The average clock frequency when the DF overlay is inserted drops by 1.25% on average compared to when the Base overlay is inserted, and 1.54% when the CDF overlay is inserted. Some of the impact on Fmax will occur if the data-override multiplexers fall on any critical paths, but most of the impact on Fmax is due to increased stress in the routing. In Figure 7, the functional override instrumentation is 330 ALMs for DF, and 355 ALMs for CDF (the conditional variant). The overhead here is due to the higher bit widths that must be supported depending on the variable type (8 to 64 bits for our benchmarks). The increase between the DF and CDF variants is due to the extra logic from the conditional unit required to drive the read_enable port on the memories as in Figure 6b. As the number of selected variables increases, the data override instrumentation area overhead increases linearly.

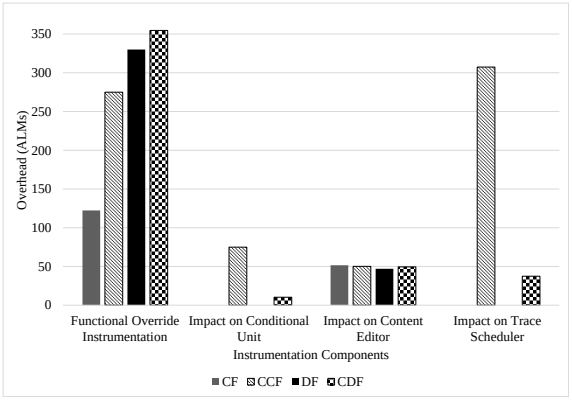



Fig. 7: Impact of Control and Data Flow Architectures on Area

## VII. Discussion: Circuit Optimizations

Throughout this work, our goal has been to provide a software-like debug experience. Ideally, the user can interact with the source code as software without worrying about the specifics of the underlying hardware implementation. If possible, this may open the door for software engineers, but also improve the productivity of hardware designers by raising the abstraction with which they view their design.

One of the challenges is related to the optimizations performed by the HLS tool. We differentiate between two classes of optimizations: compiler optimizations and back-end optimizations. Compiler optimizations include loop unrolling, function inlining, common subexpression, and dead code elimination and is typically performed on the internal representation of the source code. On the other hand, back-end optimizations are performed as the hardware is generated and mostly focus on extracting any parallelism available from the design during the scheduling algorithm.

Several of the design decisions described in Section V were motivated by the presence of scheduling optimizations. For example, the restriction that the support variables in a replacement condition must be selected from the variables in the original condition assures that scheduling optimizations do not lead to comparisons using stale values. However, our approach does *not* guarantee a software-like view in all cases. For example, common-subexpression elimination, in which assignments to variables are restructured, may mean that changes to variables using our architecture do not propagate as expected. In this way, we fall somewhat short of our goal of providing a strictly software-like debug experience. Debugging optimized software is notoriously difficult, and is subject to ongoing work. In the meantime, we provide "hardware-oriented" debug information such as a cycle-accurate Gantt chart to help expert designers understand their optimized code.

## VIII. Conclusions

In this paper, we present a flexible debug overlay that both increases the observability into an HLS-generated circuit, and supports functional changes to the design. The overlay is constructed at compile time with the user circuit, and can be configured at runtime to realize a set of debug scenarios without requiring a recompile. The overlay architectures presented in this paper allow the user to alter the control flow of their design as it executes on chip, as well as override selected variable assignments to enhance their debugging capabilities. Importantly, the overlay framework leverages information from the HLS tool such that the user need only interact with their source code which provides them with an understanding of how their software affects the hardware. This paper presents a variety of overlay variants and quantifies how each capability affects the overhead. While some variants are more expensive, we believe the capabilities presented are important in supporting an HLS designer debug and understand their design.

## Acknowledgment

We would like to thank Intel for their Intel Strategic Research Alliance (ISRA) grant that funded this research.

## References

[1] Altera, *Quartus Prime Pro Edition Handbook*, November 2015, vol. 3, ch. 9: Design Debugging Using the SignalTap II Logic Analyzer.

[2] Xilinx, *ChipScope Pro Software and Cores: User Guide*, Oct 2012.

[3] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the Sea Cucumber synthesizing compiler," in *Symposium on Field-Programmable Custom Computing Machines.*, April 2003, pp. 228–237.

[4] N. Calagar, S. Brown, and J. Anderson, "Source-level Debugging for FPGA High-Level Synthesis," in *Int'l Conf. on Field Programmable Logic and Applications*, Sept 2014.

[5] J. Goeders and S. Wilton, "Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 83–96, Jan 2017.

[6] A. Jamal, J. Goeders, and S. Wilton, "Architecture exploration for HLS-Oriented FPGA debug overlays," in *Int'l Symposium. on Field-Programmable Gate Arrays*, Feb 2018.

[7] M. Fujita and Y. Kukimoto, "Patching method for lookup-table type FPGA's." in *Int'l Conf. on Field-Programmable Logic and Applications*, Sept 1992.

[8] S. Jo, "Rectification of advanced microprocessors without changing routing on fpgas (poster)," in *Int'l Symposium. on Field-Programmable Gate Arrays*, Feb 2013.

[9] M. Fujita, "Methods for automatic design error correction in sequential circuits," in *European Conference on Design Automation with the European Event in ASIC Design*, 1993.

[10] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated Design Debugging With Maximum Satisfiability," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 11, pp. 1804–1817, Nov. 2010.

[11] A. Kourfali and D. Stroobandt, "Efficient hardware debugging using parameterized FPGA reconfiguration," in *Int'l Parallel and Distributed Processing Symposium Workshop*, May 2016, pp. 277–282.

[12] F. Eslami and S. J. E. Wilton, "An adaptive virtual overlay for fast trigger insertion for FPGA debug," in *Int'l Conf. on Field Programmable Technology (FPT)*, Dec 2015, pp. 32–39.

[13] J. Goeders and S. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[14] A. Canis, J. Choi *et al.*, "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[15] 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*, vol. 17, pp. 242–254, 2009.

[16] B. Reagen, R. Adolf, Y. Shao, G.-Y. Wai, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Int'l Symp. on Workload Characterization*, Oct 2014, pp. 110–119.