Interactive Debugging at IP Block Interfaces in FPGAs

Marco Antonio Merlini
marco.merlini@utoronto.ca
University of Toronto
Toronto, Ontario, Canada

Isamu Poy
isamu.poy@mail.utoronto.ca
University of Toronto
Toronto, Ontario, Canada

Paul Chow
pc@eecg.utoronto.ca
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

Recent developments have shown FPGAs to be effective for data centre applications, but debugging support in that environment has not evolved correspondingly. This presents an additional barrier to widespread adoption. This work proposes Debug Governors, a new open-source debugger designed for controllability and interactive debugging that can help to locate issues across multiple FPGAs.

A Debug Governor can pause, log, drop, and/or inject data into any streaming interface. These operations enable single-stepping, unit testing, and interfacing with software. Hundreds of Debug Governors can fit in a single FPGA and, because they are transparent when inactive, can be left "dormant" in production designs.

We show how Debug Governors can be used to resolve functional problems on a real FPGA, and how they can be extended to memory-mapped protocols.

CCS CONCEPTS

• **Hardware** → *Design for testability*; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

FPGA debugging; hardware debugging; controllability; immediacy

ACM Reference Format:
Marco Antonio Merlini, Isamu Poy, and Paul Chow. 2021. Interactive Debugging at IP Block Interfaces in FPGAs. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3431920.3439305

1 INTRODUCTION

In the past decade, research has confirmed that FPGAs are worthy of use in data centres. However, the difficulty of developing for them is a significant barrier to adoption, and this is particularly true when it comes to debugging. In [22], Putnam *et. al.* argue that "hardware bugs or faults inevitably occur at scale that escape testing and functional validation," and that diagnosis "requires visibility into the state of the hardware leading up to the point of failure."

In addition to visibility, we believe that practical and efficient debugging requires immediacy [26], *i.e.*, allowing a developer to interact with their design as easily as they would objects in the physical world. A developer seldom knows where to look when first encountering a bug. Instead, they start to make guesses and to test hypotheses. As their understanding improves, they need to be able to switch focus from one part of their design to another.

This work proposes an approach to FPGA debugging that prioritizes controllability and interactivity. Our result is a simple, yet general instrumentation that is effective for single FPGAs and scales naturally to multiple FPGAs in a data centre application. This is meant to *complement* existing tools by operating at a higher level of abstraction; Debug Governors can quickly narrow down the approximate location of a problem, such as to a particular FPGA, at which point any existing low-level debugger can take over. In addition, we chose to make this new development open-source [19] to foster an ecosystem of collaboration.

This paper is organized as follows. Section 2 presents an overview of our proposed debugging tools. Section 3 discusses background and related work. Section 4 presents the hardware architecture of our debugger, and Section 5 evaluates it. Finally, Section 6 concludes the paper and discusses future work.

2 OVERVIEW

We illustrate our approach with a running example where we debug the messages from a sender (Alex) to a receiver (Brittany). To do so, we pass the stream through a *Debug Governor*, as shown in Fig. 1.

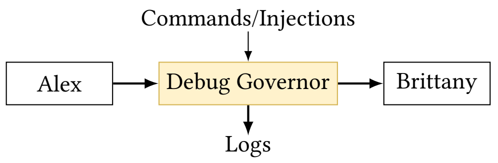

Figure 1: Data stream instrumented with a Debug Governor.

A Debug Governor can "govern" a stream of data by enabling any subset of the following four operations:

- **`Pause`:** Alex is prevented from sending data.
- **`Log`:** All data sent from Alex are duplicated and sent to the Governor's log output. The Governor will automatically pause incoming data from Alex if the receiver of the logs is not ready.
- **`Drop`:** Alex is allowed to send data into the Governor, but they are not sent to Brittany (though they can still be logged).
- **`Inject`:** Data from the Governor's inject input are sent to Brittany, pausing Alex if necessary.

We will call these the *Pause, Log, Drop, and Inject (PLDI) operations.* There are sixteen configurations for the Debug Governor, as given by the powerset of {`Pause`, `Log`, `Drop`, `Inject`}. Importantly, when no PLDI operations are active, a design with Debug Governors operates identically to one without.

Each of the PLDI operations is useful on its own. Logging allows a developer to observe intermediate values and is always needed for debugging. Pause-ing can be used to force a certain sequence of events, or to halt a high-bandwidth output that is causing congestion in a network. Dropping can be used to test a design's resilience to lost data. We place special emphasis on the Inject operation, which is rarely seen in other FPGA debugging tools. Interactive control can produce clues that are unavailable with Logging alone. For example, if a module is producing bad outputs, the correct outputs could be Injected instead to check if the rest of the design is working as intended. Alternatively, by Injecting on one Governor and Logging on another, unit tests can be performed *in situ*.

Useful debugging actions can be obtained applying several PLDI operations simultaneously. One noteworthy configuration is "single-stepping". Debug Governors include a controller that, given a number *n*, can automatically re-enable Pause-ing after exactly *n* messages are Logged. Alternatively, consider the configuration in Fig. 2, where Debug Governors are shown with the letters "DG". By Logging and Dropping on DG1, all data leaving Alex can be redirected to the input of an HDL simulation of Brittany. At the same time, by Pause-ing and Injecting on DG2, the simulation outputs can be sent to Camilo.

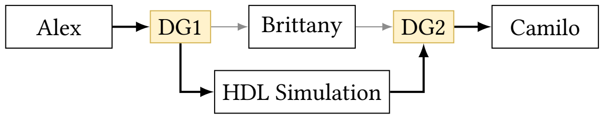

Figure 2: Bypassing a core and replacing with a software simulation.

3 BACKGROUND AND RELATED WORK

This section first provides a summary of FPGA debugging. We then describe the bus protocols used in this work. Finally, we present related work.

3.1 FPGA Debugging

A common FPGA debugging method is to add an embedded logic analyzer [15, 28] to a design. This allows a developer to see runtime values, but can only observe a limited number of signals for a short amount of time. There exist several approaches to increase an analyzer's effective capacity [7, 11, 14]. Even with these techniques, developers frequently need to change the placement of analyzers in their design, involving a long process of recompilation and potentially troubleshooting timing failures.

There are a number of alternatives to embedded logic analyzers. Most FPGAs offer a state *readback* feature that can provide 100% observability, but cannot be used to debug circuits at speed [12]. Synplify's TotalRecall [18, 21] can offer 100% visibility into circuits running at speed, but requires most or all of the original design to be instantiated twice. When using High-Level Synthesis (such as with the academic tools in [6] and [25]), it is possible to provide a source-level debugging experience [5, 10, 13] though still only for a subset of signals at a time. The Flight Data Recorder in [22] makes better use of a limited frame buffer by only saving signals of interest and not on every clock cycle.

Comparatively, few controllability tools exist for FPGAs. A developer can add special cores [16, 29] to a design, allowing them to view and/or set individual wires in their design. These "virtual switches" can be used for simple tasks such as resetting components, setting constants, or individual clock enables, but are unable to inject transactions into bus protocols. For more control, an FPGA design could be instrumented with a scan chain, but the area overhead is about 85% (compared to 5-30% for ASIC scan chains) [27]. This cost can be reduced to about 45% by combining scan chains with readback [24], or can instead be integrated into the FPGA fabric itself [23].

3.2 Handshaking and AXI Stream

Handshaking is a common idiom for managing transmission from a sender to a receiver, where both are driven by the same clock. Fig. 3 illustrates the technique, and it is described as follows. Alex drives a datum on a set of parallel lines as well as a valid bit, all of which are connected to Brittany. Brittany drives a ready bit which is connected back to Alex. Both consider the datum to have been sent if and only if both valid and ready are asserted at a clock edge. A single transmitted datum is known as a *flit*.

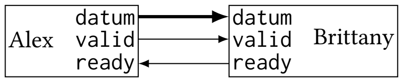

Figure 3: Handshaking.

The AXI Stream protocol [2] is a backwards-compatible extension of handshaking. Alongside the datum, valid, and ready signals (which AXI Stream renames to TDATA, TVALID, and TREADY, respectively), an AXI Stream sender may add any subset of *sidechannels* that are transmitted in parallel to TDATA. The AXI Stream specification defines a number of standard sidechannels, such as TLAST to signify end-of-packet or TDEST to specify a destination. Debug Governors specifically support AXI Stream because of its widespread use. However, AXI Stream is identical to handshaking when the (optional) sidechannels are unused. Thus, Debug Governors also support basic handshaking and are immediately applicable in this more general context.

3.2.1 *Addressing with AXI Lite.*

AXI Lite extends AXI Stream to support memory-mapped transactions. The AXI Lite protocol contains five separate AXI Stream channels: ARADDR for read address, RDATA for read data, AWADDR for write address, WDATA for write data, and BRESP for write responses. Address and data must both be provided in read/write transactions. Figure 4 shows the arrangement of these channels: each arrow represents an AXI Stream and points from the sender to the receiver *of that channel.* Note that an AXI Lite sender acts the AXI Stream receiver on the RDATA and BRESP channels.

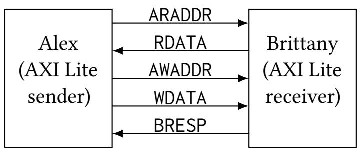

Figure 4: Arrangement of AXI Stream channels in AXI Lite.

3.3 Related Work

The Am2045 [4], a custom architecture for highly parallel computing, is a single ASIC with 336 RISC CPUs and programmable interconnect. The runtime reconfiguration capabilities of the Am2045 require the programmer to *flush* FIFOs and *halt* all transmissions until all CPUs have been programmed and started. These two operations are identical to our Drop and Pause operations (respectively), and the circuits that implement them are identical to ours. It is not clear whether the flush and hold controls in the Am2045 could be controlled individually.

Dan Gisselquist has recently published an in-FPGA instrumentation tool for observing and controlling the first input and final output of a chain of multiple AXI Stream components [9]. The developer can send arbitrary inputs into the chain and verify that the final outputs are correct. The instrumentation is only designed for use at the "top-level" inputs and outputs of a pipeline, and unintended for use as a pass-through component like our Debug Governors.

4 DESIGN

Fig. 5 shows an overview of our instrumentation. Debug Governors can be added between any sender/receiver pairs and are controlled by a daisy-chained stream of commands. These commands can selectively enable PLDI operations on any Debug Governor. All Debug Governors internally convert their Log outputs to a standard format, which are combined by an n-to-1 arbiter. Ultimately, a developer controls an entire Debug Governor network with one Command input and one Log output.

The internal structure of a Debug Governor is shown in Fig. 6. Section 4.1 describes the Handshake Governor, and details on the other blocks can be found in [20, §3]. Section 4.2 shows how the Debug Governor can be extended to memory-mapped protocols.

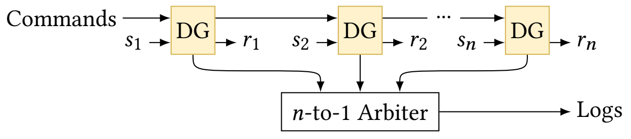

Figure 5: Complete Debug Governor instrumentation. The s_i and r_i ports mean "Sender i" and "Receiver i".

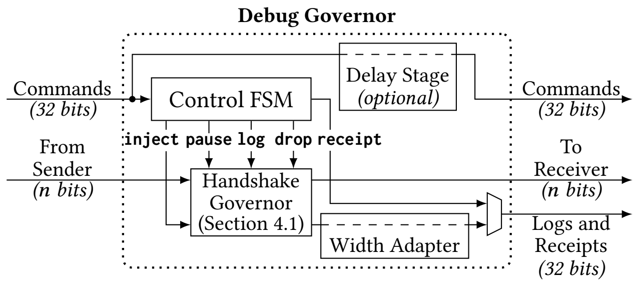

Figure 6: Top-level view of a Debug Governor.

4.1 Handshake Governor

The Handshake Governor implements the PLDI operations by manipulating valid and ready bits. It has extra ports for Injection input and Log output as well as four separate input bits to control the four debugging operations. In this section, all handshaked connections are shown with our datum, valid, and ready notation, but recall that this is forward-compatible with the AXI Stream protocol.

Fig. 7 shows how a handshaked connection can be paused. The datum line is directly connected from Alex to Brittany. If pause is low, then (Brittany_valid = Alex_valid) and (Alex_ready = Brittany_ready). In other words, the valid and ready signals are unchanged. However, if pause is high, then Brittany_valid is forced low and Alex_ready is forced low, meaning that neither will ever consider data to be sent.

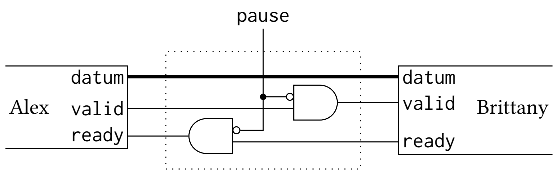

Figure 7: Pause circuit.

The dropping circuit is shown in Fig. 8. When drop is not enabled, valid and ready pass through unchanged. However, when drop is high, Alex thinks Brittany is always ready but Brittany thinks Alex is never valid.

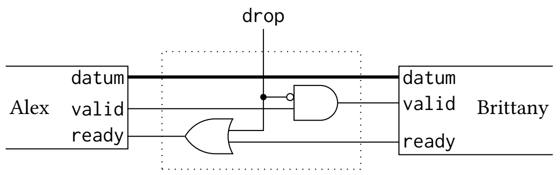

Figure 8: Drop circuit.

The injection circuit shown in Fig. 9 allows a second sender to send flits to Brittany. When Inject_valid is low, Alex_valid, Alex_datum and Brittany_ready pass through unchanged. When Inject_valid is high, Brittany instead sees the Injection data, and Alex_ready is forced low. In effect, the Inject input has higher priority than Alex.

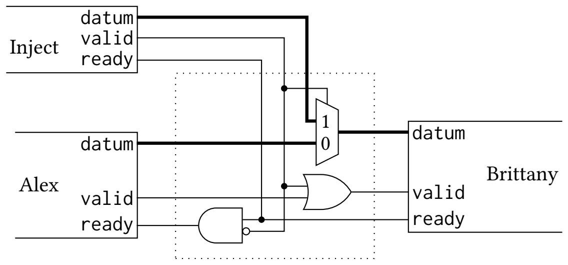

Figure 9: Inject circuit.

The logging circuit is shown in Fig. 10 and is the most subtle. A copy of the Drop circuit is used to enable/disable logging. If log_en is low, then Log_ready (as seen through the "Drop" block) appears to be high, and Log_valid is forced low.

When log_en is high, however, Log_ready and the output to Log_valid pass through the "Drop" block unchanged. Gate l_1

makes sure Alex only sends flits when *both* Brittany and the Log are ready. Gates l_2 and l_3 make sure flits are never sent to just one of the receivers; if Log_ready is low, Brittany_valid is also forced low (and vice-versa)[1].

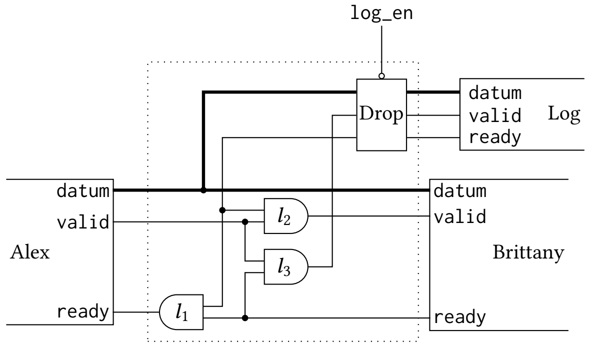

Figure 10: Log circuit.

The Handshake Governor is obtained by chaining together the circuits for each operation. However, the PLDI operations are not commutative and we must choose the order very carefully. The precise definitions of each operation, as given in Section 2, imply the following properties:

- When Pause-ing, nothing can be Logged or Dropped, but Injections can proceed.
- When Dropping, Logging and Injecting are still possible.

There are only two orderings that satisfy these properties: PLDI and LPDI. In the authors' personal opinions, "PLDI" rolls off the tongue better[2] and the final arrangement is shown in Fig. 11.

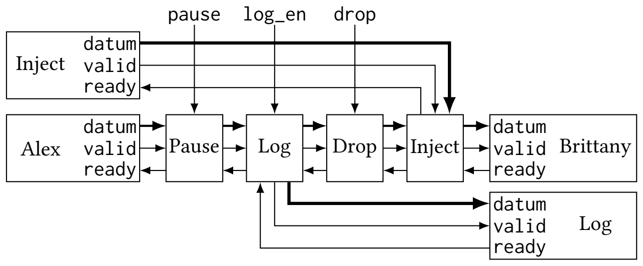

Figure 11: The Handshake Governor is obtained by chaining together the PLDI operations (in that order).

4.2 Extension to AXI Lite

Fig. 12 shows how Debug Governors can apply to AXI Lite. Each stream on the AXI Lite bus is instrumented with a Handshake Governor, and the whole is managed with an FSM. A command is structured as an Input Operation and a Data Value. The Input Operation is decoded by the FSM and used by the PLDI Enable Logic to send the proper enable signals to the Handshake Governors. The Data Value is used for specifying the value to inject on a stream.

[1]To be forward-compatible with the AXI Stream protocol, combinational paths from Brittany_ready to Brittany_valid have been eliminated.
[2]One disadvantage, however, is the slight risk of being confused with the well-known Programming Language Design and Implementation conference.

The Input Operation is used to specify Pause, Log, Drop, or Inject for one or more of the AXI Stream channels. Within the FSM, each operation results in a different sequence of states. Each operation in the FSM includes DONE and WAIT states to avoid repeated operations, and to wait for proper handshaking respectively. For example, to Inject on AWADDR and WDATA, the FSM enters a WAIT state to ensure that in-progress write transactions are completed. After finishing the Injection, the FSM de-asserts the Injectenable signal. Since this Injection mimicked a write transcation, Brittany sends a response on BRESP. This flit must be Dropped because Alex, not responsible for the write transaction, does not expect a response from Brittany.

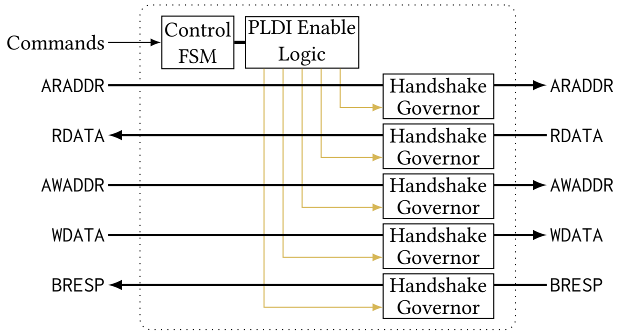

Figure 12: Debug Governor for AXI Lite.

5 EVALUATION

This section presents an evaluation of Debug Governor instrumentation. We first discuss the usability of the instrumentation by going through an example application. We then give some quantitative measurements of performance and resource costs.

5.1 Usability

This section demonstrates the usability of Debug Governors by walking through a typical debugging session. We first describe our example design; this design contains an intentional bug, and the reader is challenged to spot the mistake before we reveal it. We then provide a summary of how we were able to "find" this bug and correct it by using Debug Governors.

5.1.1 String-to-Number Converter. Our example design is shown in Fig. 13 and the source (except for `define statements) is given in Listings 1 and 2. Both modules obey the handshaking protocol described in Section 3.2.

The str_sender module repeatedly sends out an ASCII string one character at a time. The str_to_num module parses base-10 numbers that are delimited by non-digit characters. It maintains a 32-bit register named n_dtm; when an ASCII digit character is received, n_dtm is multiplied by 10 and the numerical value of the digit is added to it. When any non-digit character is received, the current value of n_dtm is output. The WAIT_FIRST_DIGIT state prevents str_to_num from outputting a number if it hasn't yet received any digit characters. Given the string in str_sender as input, str_to_num should output the repeating sequence (19, 8, 2005, 0, 5, 3759, 1597463007).

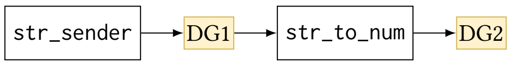

Figure 13: Example design with Debug Governors.

Listing 1: Source code for `str_sender`.

```
module str_sender(clk, s_dtm, s_vld, s_rdy);
input clk;
output [7:0] s_dtm; output s_vld; input s_rdy;
wire [35*8 -1:0] str =
    "19/08/2005: 0x5F3759DF = 1597463007";
reg [5:0] i = 0;
always @(posedge clk)
    if (s_vld && s_rdy)
        i <= (i == 6'd34) ? 0 : i + 1;
assign s_dtm = str[8*(35-i) -1 -: 8];
assign s_vld = 1;
endmodule
```

Listing 2: Source code for `str_to_num`.

```
module str_to_num(clk, s_dtm, s_vld, s_rdy,
                       n_dtm, n_vld, n_rdy);
input clk;
input      [7:0] s_dtm; input  s_vld; output s_rdy;
output reg [31:0] n_dtm; output n_vld; input  n_rdy;
reg [1:0] state = `WAIT_FIRST_DIGIT;
wire s_isdigit = (s_dtm >= "0" && s_dtm <= "9");
wire s_handshake = (s_vld && s_rdy);
always @(posedge clk)
    case (state)
    `WAIT_FIRST_DIGIT:
        if (s_handshake && s_isdigit) begin
            state <= `READ_DIGITS;
            n_dtm <= {28'b0, s_dtm[3:0]};
        end
    `READ_DIGITS:
        if (s_handshake && s_isdigit)
            n_dtm <= {n_dtm[28:0],3'b0}
                    +{n_dtm[30:0],1'b0}
                    +{28'b0,s_dtm[3:0]};
        else if (s_handshake && !s_isdigit)
            state <= `SEND_NUM;
    `SEND_NUM:
        state <= (n_vld && n_rdy) ?
                 `WAIT_FIRST_DIGIT : `SEND_NUM;
    endcase
assign n_vld = (state == `SEND_NUM);
assign s_rdy = (state != `SEND_NUM);
endmodule
```

5.1.2 Single-stepping the String to Number Converter. For this experiment, the instrumented string-to-number example from Fig. 13 was loaded onto an FPGA and connected to our user interface over TCP. Due to space limitations, we only provide a summary of the debugging session[3], but a more detailed example is given in [20, § 3.5.1].

[3]A screencast of this debugging session is available at https://asciinema.org/a/JibDBSLxdRSKwACnGNSNhW3VV. Please scroll down on the linked page for viewing information.

Table 1: Discovering a bug in the example.

Action	Results/Comments
Enable stepping on DG2.	DG2 will Log a flit every time we single-step. DG1 is still inactive and allows flits to pass normally.
Single-step DG2.	DG2 logs the value 19, as expected.
Single-step DG2 five times.	DG2 logs the values 8, 2005, 0, 5, and 3759, as expected.
Single-step DG2.	DG2 logs the value 832510767. This is different from the expected value of 1597463007.
Single-step DG2.	DG2 logs 8. The expected value of 19 was not seen.

Table 2: Understanding the bug.

Action	Results/Comments
Enable stepping on DG1.	DG1 will Log a flit every time we single-step. DG2 is still in stepping mode.
Turn off stepping and turn on Logging on DG2.	DG2 was trying to output a value but was Paused because we hadn't stepped it yet. As soon as stepping is disabled, it immediately outputs 2005.
Single-step DG1.	DG1 logs an ASCII ' ' (space) character.
Single-step DG1.	DG1 logs a '0' character.
Single-step DG1.	DG1 logs an 'x' character and DG2 logs a value of 0.
Single-step DG1.	DG1 logs a '5' character.
Single-step DG1.	DG1 logs an 'F' character and DG2 logs a value of 5.
Single-step DG1 five times.	DG1 logs '3759D'. On the fifth step, DG2 logs 3759.
Single-step DG1 four times.	DG1 logs 'F = ' (which contains no digits).
Single-step DG1 ten times.	DG1 logs '1597463007', which is the correct number from the end of the string.
Single-step DG1 two times.	DG1 logs '19'. DG2 *does not* log anything. The '19' appears to be part of '159746300719'.
Single-step DG1.	DG1 logs '/'. DG2 logs '832510767'.

Table 3: Testing the solution.

Action	Results/Comments
Single-step DG1 thirty-two times.	DG1 logs '08/2005: 0x5F3759DF = 1597463007'. DG2 logs 8, 2005, 0, 5, and 3795.
Inject a ' ' on DG1.	DG2 logs the value 1597463007. This is correct!
Single-step DG1 three times.	DG1 logs '19/'. DG2 logs 19. Also correct!

Table 1 single-steps through DG2 and uncovers two problems: in the second-last row, we saw the value 832510767 instead of 1597463007 and in the last row we saw 8 instead of 19. Table 2 shows how we found the bug. The first two rows describe how the Governors were configured. The next seven rows single-step through DG1 and show correct behaviour. The last three rows show the root cause: because the string in `str_sender` is looped, the 19 at the beginning occurs immediately after the 1597463007 at the end and `str_to_num` treats it as one big number (that also overflows 32-bit precision). The solution is to add a non-digit character at the end of the string in `str_sender`. With Debug Governors, we can confirm this solution is correct *without generating a new bitstream* by Injecting a character into DG1 at the right place. Table 3 demonstrates the results of this proposed solution.

5.2 Performance

When the Debug Governor is unused, it behaves like direct wires between Alex and Brittany. The added combinational delay is entirely due to the Handshake Governor, as seen in Fig. 6. We implemented the Handshake Governor as two parallel circuits: first, Alex_datum connects to Brittany_datum through a single multiplexer (inside the Inject block, Fig. 9); second, the remaining logic manages the handshaking signals. This second circuit has seven inputs (Alex_valid, Inject_valid, Brittany_ready, Log_ready, pause, log_en, and drop) and four outputs (Alex_ready, Inject_ready, Brittany_valid, and Log_valid). However, none of the outputs depend on more than six of the inputs. On an FPGA with six-input LUTs, the added combinational delay from Alex to Brittany is only a single LUT!

Now consider the case when the Debug Governor is active. Injecting a flit requires several commands to program the Inject control registers, and the Governor must be periodically polled to determine when the Injection has succeeded. Given these constraints, we have found that binary files can be Injected at a rate of approximately 1 KB per second. Logging, on the other hand, is only slowed down by the Width Adapter shown in Fig. 6, which sends a log and its associated metadata using several 32-bit flits. We have found that logged flits can be transferred at about 55 KB per second.

5.3 Resource Costs

A major design goal for Debug Governor instrumentation is to enable broad instrumentation of all streams in a design. In so doing, the developer would never need to edit which streams are instrumented and recompile their bitstream. This could easily mean hundreds of Governors are needed on a single FPGA. For this reason, the resource costs of each Debug Governor have been reduced as much as possible, and the implementation contains many special cases to further reduce costs. In particular, no Block RAMs are used. This section presents the number of Lookup Tables (LUTs) and Flip Flops (FFs) used by the Debug Governors and the n-to-1 arbiter.

The cost of a Debug Governor is determined by the sum of the bit widths in TDATA, TDEST, and TID, which we denote as w. If the underlying stream does not include TKEEP, a single Governor costs about $w + 89$ LUTs and $3w + 72$ FFs, and a plot is shown in Fig. 14. These formulas become $1.2w + 86$ LUTs and $3.33w + 70$ FFs when TKEEP is included.

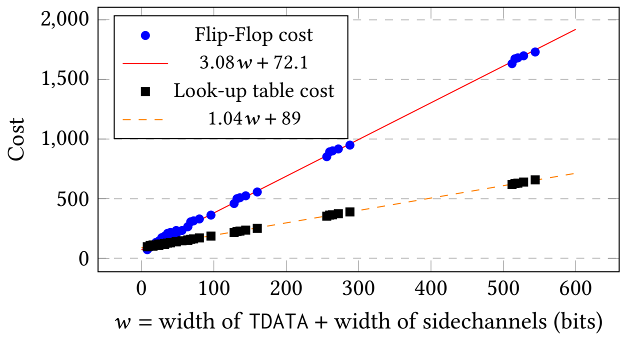

Figure 14: Debug Governor costs without TKEEP, as reported by Vivado 2018.3 Synthesis.

Consider the following example where a design is instrumented with thirty Governors, each with $w = 64$ and including TKEEP. We find that the Governors themselves consume about 5000 LUTs and 8500 FFs. Furthermore, a 30-to-1 arbiter is needed, which uses an additional 690 LUTs and 730 FFs. The totals are presented in Table 4 as percentages of resources on several different FPGA boards; note that the Zedboard is a significantly smaller FPGA than the others.

Table 4: Debug Governor Instrumentation Resource Costs, as Percentages.

Board	FPGA Family	LUT %	FF %
Alpha Data 8K5 [1]	Kintex Ultrascale	0.9	0.7
Fidus Sidewinder [8]	Zynq Ultrascale+	1.1	0.9
BEEcube BEE4 [30]	Virtex-6	1.7	1.3
Zedboard [3]	Zynq 7000	10.7	8.7

6 CONCLUSIONS AND FUTURE WORK

Our work resulted from an observation that an effective debugger requires two traits: controllability and interactivity. When consulting research colleagues and industry professionals, we also noticed a significant interest in performance debugging. Existing FPGA debuggers offer some performance debugging, but little to no controllability. We specifically addressed this in our design. We believe the result forms a basis for building the tools needed to debug the multi-FPGA applications that will be run in data centers.

It is possible to extend Debug Governors to support performance debugging. In future work we may add a transparent Logging mode that never applies backpressure, but may not log every single flit. Another extension could be a Logging mode that simply counts flits instead of Logging actual data. This is completely transparent, and still allows for useful conclusions. For example, with flit counters on either end of a FIFO, the developer can see a live view of its occupancy. Alternatively, by knowing how many flits should be passing by a certain point, the developer can observe if a particular module incorrectly drops flits.

It has also come to our attention that triggering should be supported in hardware. Currently, we can perform triggering in software by single-stepping and examining each flit until a condition is met. Instead, it would be better to eliminate the slow dialogue with a lab machine and instead extend the Control FSM.

Finally, Section 3.2.1 shows a simple example of building more complex tools from multiple Governors. Although this section focuses on the AXI Lite protocol, we can apply the ideas to support other commonly-used memory-mapped buses, such as Avalon [17]. By allowing for modular debugging, we may develop custom operations such as simultaneously Dropping read transactions while Injecting, a useful action for prioritizing debug commands.

ACKNOWLEDGMENTS

The authors would like to thank the Edward S. Rogers Sr. Department of Electrical and Computer Engineering, NSERC, and Alibaba for their financial support. Alibaba also provided helpful feedback along the way. Finally, we thank Xilinx and Fidus Systems for donating chips, tools, and support and the reviewers for their valuable comments and feedback.

REFERENCES

[1] Alpha Data. 2019. ADM-PCIE-8K5. https://www.alpha-data.com/pdfs/adm-pcie-8k5.pdf (Product Brief).

[2] ARM 2019. *AMBA® AXI4-Stream Protocol Specification*. ARM. https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf Version 1.0.

[3] Avnet Services. [n.d.]. ZedBoard. http://zedboard.org/product/zedboard (Product Brief).

[4] M. Butts, A. M. Jones, and P. Wasson. 2007. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. 55–64.

[5] Nazanin Calagar, Stephen D Brown, and Jason H Anderson. 2014. Source-level Debugging for FPGA High-Level Synthesis. In *2014 24th international conference on field programmable logic and applications (FPL)*. IEEE, 1–8.

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 33–36.

[7] EXOSTIV. 2020. Analyze, Verify and Debug FPGA: Gigabyte-Range Visibility into the FPGA at Full Speed. https://www.exostivlabs.com/exostiv/ (Product Brief).

[8] Fidus. 2018. Sidewinder-100 Datasheet. https://fidus.com/wp-content/uploads/2019/01/Sidewinder_Data_Sheet.pdf

[9] Dan Gisselquist. 2020. Debugging AXI Streams. https://zipcpu.com/dsp/2020/04/20/axil2axis.html

[10] Jeffrey Goeders and Steven JE Wilton. 2014. Effective FPGA Debug for High-Level Synthesis Generated Circuits. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[11] Jeffrey Goeders and Steven JE Wilton. 2016. Signal-tracing Techniques for In-system FPGA Debugging of High-Level Synthesis Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 1 (2016), 83–96.

[12] DINI Group. [n.d.]. Realtime FPGA Readback for Debug. https://www.dinigroup.com/web/DN_Readbacker.php (Product Brief).

[13] K Scott Hemmert, Justin L Tripp, Brad L Hutchings, and Preston A Jackson. 2003. Source Level Debugger for the Sea Cucumber Synthesizing Compiler. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003*. IEEE, 228–237.

[14] Daniel Holanda Noronha, Ruizhe Zhao, Jeff Goeders, Wayne Luk, and Steven J.E. Wilton. 2019. On-Chip FPGA Debug Instrumentation for Machine Learning Applications. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 110–115. https://doi.org/10.1145/3289602.3293922

[15] Intel. [n.d.]. *SignalTap User's Guide*. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/signal.pdf

[16] Intel. 2019. *Intel® Quartus® Prime Pro Edition User Guide*. https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-debug.pdf See §6, Design Debugging Using In-System Sources and Probes.

[17] Intel. 2020. *Avalon Interface Specifications*. Technical Report. Intel. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

[18] Mario Larouche. 2007. How To achieve 100% visibility with FPGA-based ASIC prototypes running at real-time speeds. *EETimes* (2007). https://www.eetimes.com/how-to-achieve-100-visibility-with-fpga-based-asic-prototypes-running-at-real-time-speeds/

[19] Marco Antonio Merlini. 2020. Debug Governor Source Code. https://github.com/esophagus-now/ye_olde_verilogge/tree/main/dbg_guv

[20] Marco Antonio Merlini. 2020. *Practical Debug for Dataflow Computations on One or More FPGAs*. Master's thesis. University of Toronto.

[21] Chun Kit Ng and Kenneth S Mcelvain. 2005. Method and System for Debugging using Replicated Logic.

[22] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press, 13–24. https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/ Selected as an IEEE Micro TopPick.

[23] Michel Renovell, Penelope Faure, Jean Michel Portal, Joan Figueras, and Yervant Zorian. 2001. IS-FPGA: A New Symmetric FPGA Architecture with Implicit Scan. In *Proceedings International Test Conference 2001 (Cat. No. 01CH37260)*. IEEE, 924–931.

[24] Anurag Tiwari and Karen A Tomko. 2003. Scan-chain Based Watch-points for Efficient Run-time Debugging and Verification of FPGA Designs. In *Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference, 2003*. IEEE, 705–711.

[25] Justin L Tripp, Preston A Jackson, and Brad L Hutchings. 2002. Sea Cucumber: A Synthesizing Compiler for FPGAs. In *International Conference on Field Programmable Logic and Applications*. Springer, 875–885.

[26] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (April 1997), 38–43. https://doi.org/10.1145/248448.248457

[27] Timothy Wheeler, Paul Graham, Brent Nelson, and Brad Hutchings. 2001. Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification. In *International Conference on Field Programmable Logic and Applications*. Springer, 483–492.

[28] Xilinx. [n.d.]. Integrated Logic Analyzer (ILA). https://www.xilinx.com/products/intellectual-property/ila.html (Product Brief).

[29] Xilinx. [n.d.]. Virtual Input/Output (VIO). https://www.xilinx.com/products/intellectual-property/vio.html (Product Brief).

[30] Xilinx 2015. *Virtex-6 Family Overview*. Xilinx. https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf