EasyChair Preprint

№ 1681

Contentious Live-Tracing as Debugging Approach on FPGAs

Christopher Blochwitz, Raphael Klink, Jan Moritz Joseph and Thilo Pionteck

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

October 16, 2019

Contentious Live-Tracing as Debugging Approach on FPGAs

Christopher Blochwitz, Raphael Klink
Universität zu Lübeck
Institute of Computer Engineering
23562 Lübeck, Germany
Email: {blochwitz}@iti.uni-luebeck.de

Jan Moritz Joseph, Thilo Pionteck
Otto-von-Guericke-university Magdeburg
Institute for Information Technology and Communications
39106 Magdeburg, Germany
Email: {jan.joseph, thilo.pionteck}@ovgu.de

***Abstract*—This work presents a new approach for monitoring and debugging RTL logic on FPGAs—Live-Tracing-Logic. The design combines the two most common approaches for debugging RTL logic, Scan-Chains and Trace-Buffers, while avoiding their disadvantage: First, slow and clock-controlled scans of the Scan-Chains, second, a limited time period for tracing of Trace-Buffers, respectively. The Live-Tracing-Logic connects trace-buffer modules serially, monitors signal events continuously, transmits the collected data to the host system via a high bandwidth PCIe interface, and converts the data into a VCD file. Furthermore, an automatic tool flow is introduced, which requires only two user interactions: First, using pragmas, second, starting a TCL script. The Live-Tracing-Logic is evaluated for different workloads and different tracing modes. The results show that the architecture has the capacity to continuously trace up to 3.10 GB/s of data and is only limited by the PCIe interface. Furthermore, the Live-Tracing-Logic is suitable for multi clock designs and utilizes up to 70 % less resources in comparison to the Integrated Logic Analyzer of Xilinx.**

Keywords*—*Debugging, Tracing, Logic Analyzer, Hardware, Field-Programmable Gate Array

I. Introduction

Nowadays, hardware developers have to cope with designs which are increasingly demanding, and, furthermore, the correctness of these designs must be proved. With simple designs, this can still be ensured through simulation or intuitive test scenarios. But for more complex systems, automatic tests, automatic debugging, and verification are indispensable at all stages of development.

Testing and debugging is not always possible in simulation environments, in particular when the system consists of several units, whether software – hardware or hardware – hardware, which interact which each other and must be tested. It is also possible that the design is so complex [1] that the effective simulation frequency drops so much that a debugging is not even possible for time reasons.

For testing and hardware debugging, additional elements are required, which collect data and evaluate the functionality of the hardware design. Since it is not possible to verify all functions of a hardware design on its I/O pins, this must be done within the design/on-chip. FPGA vendors, therefore, provide so-called Integrated Logic Analyzers (ILA) [2], which require a relatively small effort to check/debug a hardware design. These Logic Analyzers utilize resources of the FPGA and should be applied accordingly. Further, only a certain number of clock cycles can be observed and stored for a defined period of time. The period corresponds with the size of the used memory and its maximum is 131072 entries for Xilinxs FPGAs [3]. The monitoring of such a time period is activated by configurable triggers. If the on-chip memory is completely filled with tracing data, it must be read out via an external interface. After reading out the memory, the signal can be observed again. Typically, the external interface is JTAG, which is very slow in relation to the high volume of data produced by the hardware design. Hence, during the time of the memory readout, states can not be observed which may be of interest for the developer.

This paper presents a design that logs data continuously and forwards it off-chip to a host system. Any number of signals can be observed and there is no limitation of the length of the observation time. The recorded data are forwarded on a FIFO-based chain to any possible external interface, which also only needs a FIFO interface. To achieve high performance, we used PCIe. In contrast to other approaches, it is possible to trace continuously a huge number of signal changes, without reducing or controlling the user's design clock. The modules can be parameterized at runtime, which increases the flexibility and makes a new synthesis unnecessary. The data are then received by a host system and converted to the VCD (Value Change Dump) [4] format, which can be displayed by all common Waveform viewers. The whole tool flow is automatized and only the signals of interest must be specified before starting a TCL-script.

The rest of this paper is structured as follows. Section II presents related works of generic and application-specific approaches. In Section III, the Live-Tracing system and its components are presented. The tool flow and the use of the system is given in Section IV. Afterwards, the design is evaluated (Section V) for different design sizes and its utilization is compared to the integrated solution provided by Xilinx (Section VI). This paper is concluded in Section VII.

II. Related Work

Debugging is still challenging and requires many additional tools, hardware, and knowledge. Debugging RTL logic on an FPGA is a special case because it allows to integrate debugging and tracing units on-chip. In most cases, the FPGA has an interface (JTAG, UART), which is indented for debugging and tracing. These interfaces also mostly allow a software tracing

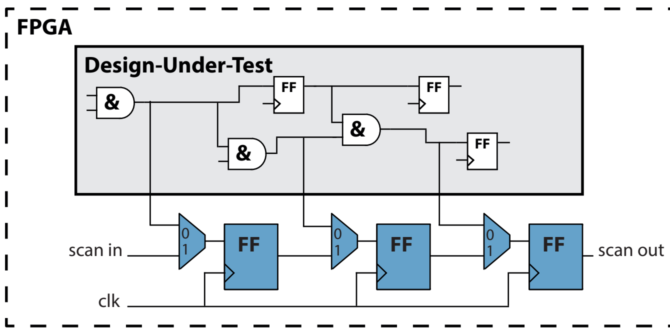

Fig. 1. Scan-Chain beside the DUT with its atomic element, the Scan-Cell

for integrated processors as the ARM Cortex or softcores (e.g. Xilinx Microblaze). However, they are only off-chip interfaces and the actual logic must be implemented by the developer. In general, there are three different main approaches: first, the Scan-Chain (Figure 1) consisting of so called scan cells—a single pair of a multiplexer and a FlipFlop—, which are connected in a chain to an external port. Basically, the chain is a large shift register, where the input can be driven by the previous cell or a debugging signal. The scan or debugging is controlled by a software enabling and disabling the user's design clock, reading out the data, enabling the clock, and so on. Especially for large designs, the approach is very slow because all data must be shifted in the chain and the time region of interest might be on a special signal combination, which is very rare. Trace Buffers are used for the second approach, signals are connected to an on-chip memory and, after an event, the signal is captured for a specific number of clock cycles, the so called window. The advantage of trace buffers are: the clock frequency is not controlled by the debugging system, which allows debugging interfaces and protocols, which are not able to reduce the clock. Second, the start of the windows can be defined by specific events, so called triggers. After the window is filled, by capturing the value of a signal for a given number of clock cycles, the data can be read-out from the debugging interface. Obviously, the limited window size reduces the significance of the trace and a bug can possibly not be detected. If larger windows or more signals are needed, the ILA uses more on-chip memory which results in a high utilization. Most of the vendors like Xilinx [2] and Altera [5] use trace buffers in their debugging tool flow. The third approach is hybrid and combines the scan chain controlling and the trace buffers with an additional external hardware logic analyzer. On-chip trace buffers are used, which trace signals in parallel and transfer the data to the external logic analyzer, which, on the other hand, controls the trace buffers and the clocks.

There are also different research works for debugging FPGAs or System-On-Chips with an integrated programmable logic. Goeders and Wilton [6] and Pinilla and Wilton [7] introduce on-chip debugging systems for high level synthesis. The approach has the ability to remap the variable of the high level language to its counterpart signal on the FPGA, hence, a convertible solution for HLS debugging. The dynamic tracing architecture of Goeders et al. recorded up to 127 times more signals in comparison to a classical embedded logic analyzer. Monson and Hutchings integrated so called *Event Observability Ports*, which enable the tracing only if a signal change is detected on this specific port. They also used their system for debugging high level systems and capture 2.0 up to 3.88 more events than a standard embedded logic analyzer.

A different approach for reducing the debugging time of the developer was introduced by Hung and Wilton [8]. They integrated incremental trace buffers for every signal which was supposed to be traced after one debugging cycle. This method needs no re-synthesis of the circuit and trace buffers inserted by placing and routing on free resources of the FPGA. Hence, the approach allows up to 98 times faster debugging cycles and makes the debugging more efficient. An overlay debugging architecture was published by Eslami and Wilson [9], which is integrated after the user design is placed and routed. The overlay uses free resources of the FPGA and triggers can be reconfigured on debugging time, which allows a more flexible use of the debugging system and no additional synthesis is needed. Jassi et al. [10] debugs System-On-Chips with hardware-software-co designs automatically by introducing graph grammars and the insertion of vendor specific intellectual properties for monitoring the RTL logic. The debugging system was proved by a video streaming application based on an AXI interconnect.

A brute force approach is used by Kourfali and Stroobandt [11], which connects all signals to a large multiplex network on routing resources after the place and route of the user design is finished. At the endpoint of the multiplex network, a tracing logic is placed, which traces and transfers the logic to an external debugging system. The multiplex network is controlled by the debugging system by using reconfiguration and allowing to select the signals of interest in the debugging time. The debugging overhead to reconfigure the multiplex network pays back after 5000 debugging cycles, but the area overhead is very small so it is also possible for large designs with a high utilization of the FPGA. A similar approach is introduced by Panjkov et al [12], who also use a multiplex network for all signals of a design. Contrary to the previous approach, the multiplex network is connected to the tracing logic of the Xilinx Integrated Logic Analyzer, which allows a more comfortable and flexible use of the vendor's tools and scripts.

The start–stop approach of ul Hasan Khan and Göhringer [13] combines the selection of the signals of interest, trace buffers, and a clock controlling. It achieves a similar utilization compared to Xilinx ILA and is not limited in window size. Also, Panjkov et al. [14] use a start–stop approach, which is compared to a scan-based and a trace-buffer system of Xilinx. The results show that the overall debugging time can be reduced, because fewer synthesis are needed in comparison to ILA's trace-buffers respectively the scan time is much smaller than the chain-scan. In this paper a different approach—Continuous Live-Tracing—is presented, which enables new possibilities for debugging RTL logic and is also much faster. The Live-Tracing design allows a continuous tracing by only transmiting events, where the signal values change and a true trigger condition are detected. The high bandwidth tracing is realized by an internal *Tracing-Logic* and an external bus interface (e.g. PCIe) to a host system, which do not require a controllable clock to trace all the required data. Hence, the *Tracing-Logic* allows to debug interfaces or designs, which requires a high and stable clock over a long time period.

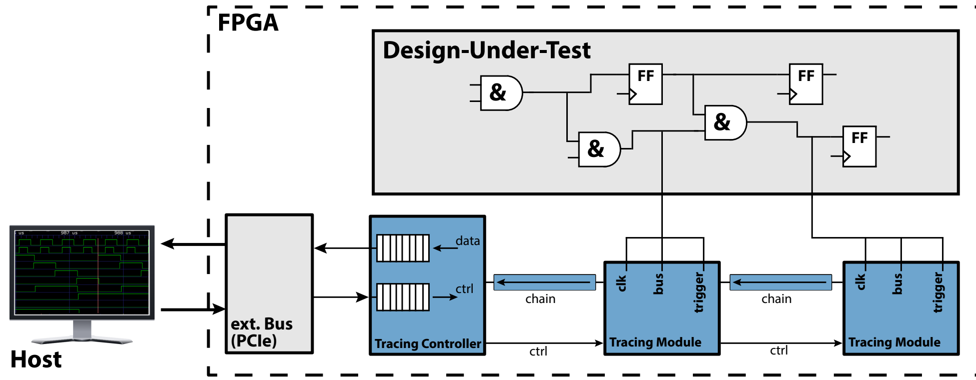

Fig. 2. Overview of the *Live-Tracing-Logic* and its integration into the Device-Under-Test. Furthermore, the external bus interface and the host are shown.

III. System Overview

The Live-Tracing design allows a continuous tracing of signal events with a high bandwidth, which is presented in this section. It consists of four main parts, which can be seen in Figure 2: First, the Design-Under-Test (DUT), which is the user's design to debug. Second, the Live-Tracing-Logic which is connected to signals of the DUT and captures the data. Third, the host system to which the FPGA is connected and where the data is converted into a VCD file, and at last, an any bus interface between the Live-Tracing-Logic and the host. In our case, we use the PCIe IP core Xillybus [15] with a maximum bandwidth of 3.5 GB/s for each direction.

The debugging system must adapt different requirements of the developer and, therefore, a flexible solution is needed. Hence, the Live-Tracing-Logic is structured into two module types: The *Tracing-Module*, which is connected to a single debugged signal of the DUT. Multiple *Tracing-Modules* are connected to each other via a parameterizable chain and at the end, to the *Tracing-Controller*, which is the connector for the bus interface (Figure 2). Furthermore, there are some control signals to the *Tracing-Modules* to configure them on debugging time. The communication between the chain's member and the bus interface is realized by a packet-based protocol. This allows a high bandwidth without arbitration time for the bus.

A. Tracing-Module

The Tracing-Module is one of the two main units of the Live-Tracing system. The module (Figure 3) is connected in a chain between two other *Tracing-Modules* (except for the first and last module) and, therefore, it has a *chain input* (connected to the *Chain-FIFO*) and *output* port. The module has unique identifier, which is required for identifying the source of the trace data. The *data* port is connected from the debugged signal to a FIFO (hereinafter referred to as *Data-FIFO*) with a corresponding width. Both FIFO depths are parameterizable: the *Data-FIFO* can be customized for every instance of the *Tracing-Module* and the *Chain-FIFO* depth is the same for all instances. The *Data-FIFO* has a clock-independent interface, which allows to trace signals with a different clock than the chain clock. Furthermore, the *Data-FIFO* is asymmetric to adapts the width-parameter of the chain. The *write_enable*-Port of the FIFO is connected to none, one, or multiple trigger signals of the DUT. Every time the trigger logic is high, a snapshot is taken by setting the *write_enable* high for one clock cycle with an additional time-stamp. If no trigger condition is defined, the *Tracing-Module* takes a snapshot of any change on the bus. The trigger logic is configurable at runtime by the *Tracing-Controller*, which will be described in more detail in the following.

The main task of the module is to send the captured data to the *Tracing-Controller*. There are two possible actions: First, forward a packet from a previous module to the chain output or create a packet out of the individual data of the *Data-FIFO*. There are different modes to decide which of the two FIFOs is read. To switch between both FIFOs, no additional clock cycle is needed.

Tracing-Modes:

- First *Chain-FIFO* — This mode prioritizes the *Chain-FIFO* and ensures that data is received from the tail of the chain on a busy bus.
- First *Data-FIFO* — This mode prioritizes the *Data-FIFO* and ensures that data is received from the head of the chain on a busy bus.
- Round-Robin — In case both FIFOs have data, the forwarding switches between both FIFOs after every packet.
- Fill-Level — The FIFO with the higher fill level will

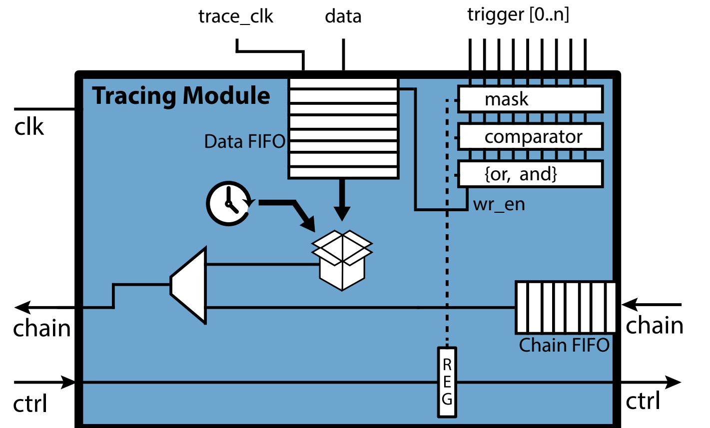

Fig. 3. Tracing-Module with the *Data-* and *Chain-FIFO*, Trigger-Logic beside the configuration registers, inner logic for package generation, and the chain.

be read, which allows to react on bursts in the DUT and balances the chain.

Runtime-Configuration: The trigger logic can be configured during runtime by sending a message from the host via the *Tracing-Controller*. Therefore, a small chain—with a single input register—from module to module is implemented. The module has an input register and an output (see Figure 3), analyzes the packet, and, if the destination ID of the packet is equal to its own, the data is written into the configuration register of the trigger logic, otherwise the packet is forwarded to the next module. The first parameter of such a configuration message is a mask of the trigger inputs, which allows to de-/activate specific triggers. The advantage of the implementation is that the developer can relatively generously define triggers and no reimplementation is needed for other triggers. Second, the value of the comparator register which is compared with the actual value of the trigger. The third parameter switches between the *or* and *and* concatenations of all trigger signals. This allows more complex trigger cases.

B. Tracing-Controller

The *Tracing-Controller* is placed between the host bus and the tracing chain. The connection to the host bus consists of two FIFOs, one for every direction. As mentioned before, the connection with the chain is also FIFO-based. The *Tracing-Controller* has two main tasks: First, forward the data of the *Chain-FIFO* to the host bus. Second, receive control-messages from the host and process of these messages. There are three types of configuration messages: First, a software reset from the host is received and all the *Tracing-Modules* are reset by a reset signal. Second, enable or disable the tracing, which can be used to reduce the data volume and only capture a smaller time window, which is the region of interest. Third, the forwarding of the trigger parameter for the *Tracing-Modules*. Therefore, a configuration packet is forwarded to the *Tracing-Modules* via the control chain.

An additional functionality for the developer is to define a maximum number of messages generated by the Live-Tracing-Logic. This allows—similar to the tracing enable—to capture only a region of interest.

C. Host-Software

The *Host-Software* is executed on the host system, where the Live-Tracing-Logic is connected with an external bus, in our case it is the Xillybus PCIe core (PCIe 3.0 8x) [15]. The software consists of several console commands for tracing and controlling the Live-Tracing-Logic. The main task of the software is to receive the tracing data from the logic and convert it to an VCD-file [4]. The Value Change Dump is a compact format, which only saves changes of a signal/value in a chronological order along with a timestamp of the change. Furthermore, a header of information such as the signal and module names and some other meta information are added. For the initializations, the software needs information about the Live-Tracing-Logic, which is parsed from an XML file. The XML file is created during the instantiation of the Live-Tracing-Logic (see Section IV). Afterwards, the data is read from a stream of the bus interface and converted to the VCD file. This can be done during the execution and allows a live view of the traced data. The mapping of the data to a traced signal is done by comparing the information of the XML file and the received ID of the packet (see also Section III-D).

As mentioned in Section III-A, the Live-Tracing-Logic can be configured at runtime. Therefore, a console command with the corresponding parameter of the module is executed and the packet is transmitted from the host software to the module via the bus interface. In the same way, a software reset is possible.

D. Chain Protocol

The communication between the host and the *Tracing-Modules* is realized by a chain and an external bus interface. Therefore, a packet-based protocol is used, which will be explained in this section in detail. The packet frame of the downstream (FPGA to host) can be seen in Figure 4 and consists of a *packet length* field, which is adapted to the widest traced bus on the chain. Further, an ID of the corresponding *Tracing-Module*, a 1-bit *Event-Flag*, a 1-bit *Overflow-Flag*, and the *Payload* is part of the packet. The length of the ID field depends on the number n of *Tracing-Modules* in the design and is calculated with $\lceil log(n) \rceil$. The payload consists of a 24-bit timestamp and the data of the traced bus with a corresponding length. In the example of Figure 4, a design with 8 modules, and a chain-width of 16-bit are used and the traced signal has a width of 56 bit. In sum, the packet has 88-bit and consists of 6 flits. The remaining bits are padded with '0' and are excluded on the host side. The *event flag* indicates a data packet with '0' and a special event with '1'. The *Overflow-Flag* is activated if the counter for the timestamp has an overflow. The meaning of the flags in combination is shown in Table I. As can be seen, the overflow of the timestamp counter generates packets regularly, which allows a parallel conversion into the VCD file (compare Section III-C).

packet length	ID	event	o-flow	timestamp
timestamp				
trace data				
trace data				
trace data				
trace data				0000 0000

Fig. 4. Packet format of the Chain Protocol.

TABLE I. DESCRIPTION OF OVERFLOW- AND EVENT-FLAG

Event-Flag	Overflow-Flag	Description
0	0	data packet with tracing data
0	1	data packet with tracing data and in the same clock cycle of capturing the bus, the timestamps overflows
1	0	any other event, which information is stored in the payload
1	1	overflow of the timestamps counter

However, there are some additional events which must be handled. For example the stop message, which indicates that all packets are transmitted after the tracing has been disabled. There is also the possibility of overfull *Data-FIFOs*. Then an error message is generated with a signal value of 'X' – unknown. Hence, the user can identify faulty traces in the VCD viewer.

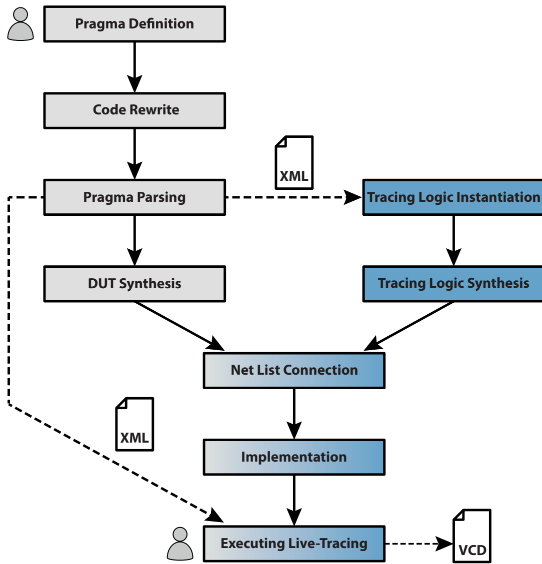

Fig. 5. Tool Flow phases, their dependencies, and file generation. Only the marked phases (person) require user interaction.

IV. Tool Flow

In this section, the tool flow from the declaration to trace a bus up to the execution will be described, which is shown in Figure 5. As mentioned in Section III, the system is flexible and optimized to minimize the influence on the tested design. Most parts of the tool flow for the integration of the Live-Tracing-Logic is automatized by using TCL scripts.

Pragma Definition: At the first step, the user adds pragmas *trace_bus()* to the code (as shown in Listing 1) at every position where a bus should be traced. The pragma has a minimum of two parameters: *Bus*, which is the traced bus, and the associated *Clock*. The *Trigger* parameter can be a list of zero up to any possible number of signals (compare Section III-A). If a more detailed configuration, especially the depth of the *Data-FIFO*, is needed, the pragma *configure_bus()* can be added to the code. This allows a fine-grained configuration for every single *Tracing-Module*.

```
-- trace_bus( Bus, Clock, Trigger*)
--
-- optional:
-- configure_bus( Bus, Data_FIFO_depth)
```

Listing 1. VHDL example als comment

Code Rewrite: The second step parses all source files for *trace_bus()* and rewrites the code by adding additional attributes for the traced bus, for example *KEEP* and *DONT_TOUCH*, which are necessary to keep the bus names for the net list.

Pragma Parsing: Third, the parameter with further information such as the bit width are parsed and saved. In addition, the information are exported in an XML-file, which is necessary to instantiate the host software.

Design Synthesis: Forth, the design is synthesized into the net list. In the fifth step, for every traced bus a *Tracing-Module* is instantiated with the parsed parameters of step three.

Live-Tracing-Logic Instantiation: The *Tracing-Controller* and all *Tracing-Modules* are connected in the sixth step and so is the connection to the external bus, which is defined as an FIFO interface.

Live-Tracing-Logic Synthesis and Net List Connection: Seventh, the Tracing-Logic is synthesized and the connection of the *Tracing-Module* with the signals of the net list is realized by TCL commands. This procedure reduces the influence on the synthesis quality and allows a custom configuration of the synthesis process.

Implementation: At this point, the Live-Tracing-Logic and the original design are merged and will be implemented for the specific FPGA.

Executing Live-Tracing: In the end, the *Live-Tracing* can be performed by executing the host software (see Section III-C).

In conclusion, the user must only do two things for Live-Tracing: Pragma definition and starting the TCL script. Hence, this tool flow is easy to use and as flexible as the integrated debugging system of Xilinx.

V. Evaluation

In this section, the Live-Tracing system is evaluated for different workloads and different modes, which has the goal to determine the limit of the Live-Tracing-Logic. As mentioned before in Section III-A, the Live-Tracing design is parameterizable and so are the FIFOs of the *Tracing-Module*. On the one hand, a larger FIFO depth means more tracing events can be saved temporally, on the other hand an increasing utilization. In this section, an optimal FIFO depth is determined for different system workloads, modes, and numbers of *Tracing-Modules*. Therefore, we implemented a test system with multiple Linear Feedback Shift Registers as pseudo number generators (see also Figure 6), which are traced by the Live-Tracing-Logic. Every pseudo number generator has an output bus with a width of 225 bits (in sum with the header 256 bits) and a 100 MHz clock. The *Tracing-Module* captures the data in the case of a value change and has no dedicated trigger implemented. The chain has a capacity of 3.2 GB/s, a width of 256 bits, and is driven by a 100 MHz clock. Furthermore, a high bandwidth PCIe bus (Xillybus) is used with a measured maximum speed of about 3.38 GB/s in each direction. For a variation of the workload, a configurable Injection-Rate is implemented for every number generators. An Injection-Rate of 10 % means that in every 10th clock cycle, the number on the bus is changed and the value is captured. The injection is equally distributed over all *Tracing-Modules* and, therefore, the capacity of the whole system is tested. In the case that the capacity of the Live-Tracing-Logic or a single *Tracing-Module* is reached, a *Data-FIFO* overfull event (hereinafter X-event) is detected and queued to be sent it to the host system. This is the case when the *Tracing-Module* cannot empty the *Data-FIFO* because of an accumulation of the chain or if the traced signal triggers more events (e.g. a long burst) than are readable by the *Tracing-Module* or bufferable by the *Data-FIFO*. Because of the fluctuation of the PCIe interface, the driver, and the host software, all tests are repeated five times and have a relative standard deviation of 0.011 %. The deviation of PCIe is caused by the serial protocol of the interface, where data

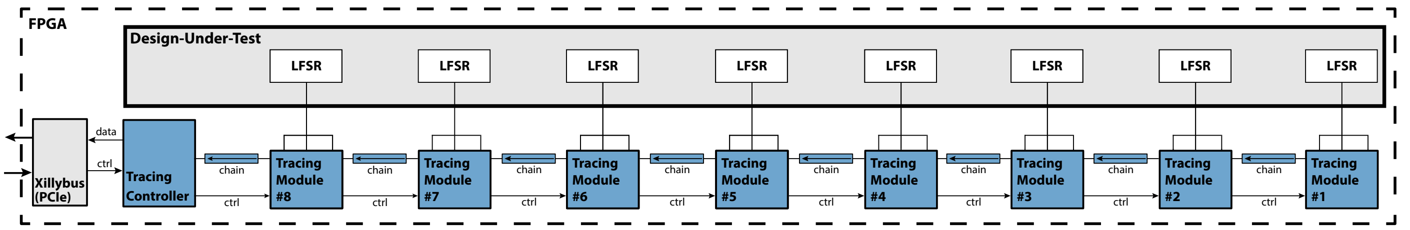

Fig. 6. Evaluation design consisting of 8 LFSRs as random number generator, the corresponding *Tracing-Modules*, and the *Tracing-Controller*.

TABLE II. FILL LEVEL OF DATA AND CHAIN FIFO IN RELATION TO INJECTION RATE. {FULLEST MODE, 512 FIFO DEPTH}

injection	module #8		module #7		module #6		module #5		module #4		module #3		module #2		module #1	
rate [%]	data	chain	data	chain	data	chain	data	chain	data	chain	data	chain	data	chain	data	chain
6	1	3	1	3	1	2	1	2	1	2	1	2	1	2	1	0
7	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	0
8	1	3	1	3	1	2	1	2	1	2	1	2	1	1	1	0
9	1	3	1	3	1	3	1	2	1	2	1	2	1	2	1	0
10	1	3	1	3	1	2	1	2	1	2	1	1	1	1	1	0
11	28	78	1	3	1	2	1	2	1	2	1	2	1	2	1	0
12	511	510	511	510	511	510	511	510	511	510	511	510	511	510	0	0
13	511	510	511	510	511	510	511	510	511	510	511	510	511	510	511	0
14	511	510	511	510	511	510	511	510	511	510	511	510	511	510	511	0
15	511	510	511	510	511	510	511	510	511	510	511	510	511	510	511	0

TABLE III. NUMBER OF THOUSAND X-EVENTS (OVERFILL DATA FIFO) AND MILLION RECEIVED MESSAGES. {FULLEST MODE, 512 FIFO DEPTH}

injection	bandwidth		module #8		module #7		module #6		module #5		module #4		module #3		module #2		module #1	
rate	theo.	meas.	X'evt	msg	X'evt	msg	X'evt	msg	X'evt	msg	X'evt	msg	X'evt	msg	X'evt	msg	X'evt	msg
[%]	[GB/s]	[GB/s]	[K]	[M]	[K]	[M]	[K]	[M]	[K]	[M]	[K]	[M]	[K]	[M]	[K]	[M]	[K]	[M]
6	1.43	1.43	0	17.9	0	17.9	0	17.9	0	17.9	0	17.9	0	17.9	0	17.9	0	17.9
7	1.67	1.67	0	20.9	0	20.9	0	20.9	0	20.9	0	20.9	0	20.9	0	20.9	0	20.9
8	1.91	1.91	0	23.9	0	23.9	0	23.9	0	23.9	0	23.9	0	23.9	0	23.9	0	23.9
9	2.15	2.15	0	26.9	0	26.9	0	26.9	0	26.9	0	26.9	0	26.9	0	26.9	0	26.9
10	2.38	2.38	0	29.9	0	29.9	0	29.9	0	29.9	0	29.9	0	29.9	0	29.9	0	29.9
11	2.62	2.62	0	32.9	0	32.9	0	32.9	0	32.9	0	32.9	0	32.9	0	32.9	0	32.9
12	2.86	2.86	0.8	35.7	0.8	35.7	0.8	35.7	1.2	35.7	1.4	35.7	9.8	35.7	6.4	35.9	0	35.9
13	3.10	3.10	3.0	38.4	2.9	38.4	2.9	38.4	4.8	38.4	11.7	38.4	211.0	38.1	135.1	38.2	0.4	38.9
14	3.34	3.08	7.3	41.0	7.4	41.0	8.3	41.0	17.6	40.9	3 169.2	37.3	10 676.3	21.2	8 968.0	26.8	1.6	41.7
15	3.58	3.04	4.1	44.1	4.2	44.1	5.6	44.1	58.1	44.1	9 464.9	31.6	11 686.4	17.8	10 303.6	13.8	4.9	44.4

and control messages share the same link. The host system running CentOS 7 and contain a 4 core Intel Xeon E3-1226 V3 with a base frequency of 3.3 GHz, 32 GB DDR3 memory, and a Xilinx Virtex Ultrascale Evaluation Board - VCU 108. The hardware design was implemented in VHDL using Xilinx Vivado 2017.1.

Every test had a length of 3 s and was controlled by the host software by enabling/disabling the tracing via the *Tracing-Controller*. The stream from FPGA to host was written into the RAM memory to ensure a maximum performance. Afterwards, the trace was processed by the host software, the VCD file was generated, and different parameters were analyzed – at first, the fill level of every FIFO of the *Tracing-Modules*, second the numbers of X-event messages because of overfull FIFOs, and third, the overall bandwidth.

In Table II and III, the fill level of the *Data-* and *Chain-FIFO* and the number of X-events for every module compared to an increasing Injection-Rate is shown. The design includes 8 number generators and 8 *Tracing-Modules* (module #8 is connected to the *Tracing-Controller*) with a fixed FIFO depth of 512 and the *fullest first mode*. As can be seen, up to an injection rate of 11 %, no X-event messages are received, which means that all data are traced successfully. The resulting bandwidth is 2.62 GB/s and all of the FIFOs their maximum fill level is up to three values (except module #1). It can also be seen in Table II that the achieved bandwidth of the PCIe is similar to the theoretical bandwidth, up to an injection rate of 12 %, which means that the system has the ability to transmit all traced data to the host system. On the other hand, there is a gap between the theoretical and the measured bandwidth for higher injection rates and furthermore the maximum of Xillybus PCIe core is not reached, which means that the host system cannot copy the trace into the RAM fast enough. As mentioned, the bandwidth for copying the trace data for the injection rate of 12 % is high enough, but there are still X-events and the FIFOs are full, which means that the system is limited by other processes. One possibility is the unsteady copying of the host system and, as a result, an overfull PCIe DMA system (64 MB). This hypothesis can be supported by analyzing the timestamps of the X-events, which are all within the first 100 ms and there are also some tests runs with a injection rate of 12 % without

TABLE IV. NUMBER OF THOUSAND X-EVENTS (OVERFILL DATA FIFO) AND MILLION RECEIVED MESSAGES IN RELATION TO THE MODES. {INJECTION_RATE 15%, 8 MODULES, 512 FIFO DEPTH}

mode	module #8		module #7		module #6		module #5		module #4		module #3		module #2		module #1	
	X'evt [K]	msg [M]	X'evt [K]	msg [M]	X'evt [K]	msg [M]	X'evt [K]	msg [M]	X'evt [K]	msg [M]	X'evt [K]	msg [M]	X'evt [K]	msg [M]	X'evt [K]	msg [M]
chain first	2 306.0	2.3	0.4	43.9	0.4	44.2	0.4	44.5	0.2	44.7	0.1	44.9	0	45.1	0	44.7
data first	0.1	44.9	0.2	44.8	0.4	44.8	0.6	44.7	0.8	44.5	0.6	43.9	2.1	43.2	2 844.2	2.9
round robin	0	44.9	0	44.9	0	44.9	2.5	44.8	21.2	44.7	111.8	44.4	10 394.8	14.3	10 578.0	14.1
fullest first	1.9	44.5	2.0	44.5	2.2	44.5	50.5	44.5	9 463.1	31.6	11 204.6	17.8	10 679.8	14.7	2.1	44.6

an X-event. Therefore, we implemented an additional design with an optimized Xillybus IP core with a DMA buffer size of 512 MB. It was demonstrated that a higher bandwidth is possible by using a larger DMA buffer and the limiting point is, up to now, the PCIe interface.

As can be seen in Table II, there is a hard cut between an injection rate of 11 % and 12 %. Hence, the limit of the system is a value of approximately 11 %, which is the last tested configuration without X-events, which also has a low memory requirement. Therefore, we recommend small FIFOs, which have a lower hardware requirements. But the configuration of the FIFOs' depths must be customized for every specific case. For example, for a traced bus with a higher clock than the chain clock, which produces data by bursts, an integration of a FIFO for buffering is indispensable.

In Table IV, the different modes are compared to each other. The results show, that there is a difference between the modes, if the Live-Tracing system is beyond its limit (injection rate of 15 %). The *data first* and the *round robin* mode prioritize the *Data-FIFO* and produce an accumulation of the chain, which can be seen at the lower message number of the *Tracing-Modules* of the tail (module #1). As expected, the *chain_first* mode prioritizes the tail of the chain and the *fullest_first* mode is more balanced with higher X-event rate in the middle of the chain.

VI. UTILIZATION

In this section, we analyze the utilization of the Live-Tracing-Logic (LUT, Flip-Flop, Block-RAM) for different configurations. The configurations are compared with ILA. For both the Live-Tracing-Logic and ILA, only the core modules are compared without the external bus interface. For the first configuration, the number of *Tracing-Modules* is varied and the parameters are fixed to: *fullest first mode*, chain and *Data-FIFO* depth of 512, and only one clock, which is shown in Table V. The ILA was integrated with the additional trigger function, which has a comparable functionality to the trigger logic of the Live-Tracing system. As expected, the utilization increases with the number of integrated modules. As can be seen, the ILA needs from 1.5 up to 3 times more LUTs, and 2.4 up to 3.3 times more Flip-Flops. However, the number of used Block-RAMs is lower for the ILA system. As mentioned in Section V, if the chain is not fully occupied, the required depths of the FIFOs are very small and, therefore, the required number of Block-RAM can be reduced significantly. Therefore, a design with FIFOs with the depth of 32 is implemented, which uses distributed memories (see also Table V, entry 8 (short)).

TABLE V. UTILIZATION OF TRACING-LOGIC COMPARED TO XILINX'S ILA FOR A NUMBER OF MODULES. {1 CLOCK, FULLEST MODE}

modules	LUT		Flip-Flop		Block-RAM	
	trace	ILA	trace	ILA	trace	ILA
total	537 600		1 075 200		1 728	
1	997	3 038	1 573	5 238	22.5	6.5
4	3 416	5 454	4 753	12 547	45	25.5
8	6 665	9 687	8 993	22 312	75	51
16	11 471	17 380	17 474	41 828	135	102
8 (short*)	8 351	-	15 651	-	0	-

* The depth is reduced to 32 and distributed memory is used.

For the second configuration, multiple clocks were integrated (Table VI) by using a system of 8 modules. The utilization of the Live-Tracing-Logic is the same for every number of clocks, that is because of the used clock-independent *Data-FIFO*, which solves the problem of synchronization. The utilization of the ILA increased with the number of clocks and is up to 1.5 times higher for LUTs and 1.4 times higher for the used Flip-Flops compared to the ILA 1-clock implementation. This increases the utilization gap between the Live-Tracing-Logic and the ILA further.

TABLE VI. UTILIZATION OF TRACING-LOGIC COMPARED TO XILINX'S ILA FOR A NUMBER OF CLOCKS. {8 MODULES, FULLEST MODE}

clk	LUT		Flip-Flop		Block-RAM	
	trace	ILA	trace	ILA	trace	ILA
total	537 600		1 075 200		1 728	
1	6 665	9 687	8 993	22 312	75	51
2	6 665	10 903	8 993	25 094	75	51
3	6 665	13 052	8 993	27 894	75	52
4	6 665	14 944	8 993	30 696	75	52

The utilization of the different modes of the *Tracing-Module* is also compared (Table VII). As can be seen the utilizations are very similar for all modes and, therefore, should not be taken into account for design decisions.

TABLE VII. UTILIZATION OF TRACING-LOGIC FOR A DIFFERENT MODES. {8 MODULES, 1 CLOCK}

mode	LUT	Flip-Flop	Block-RAM
total	537 600	1 075 200	1 728
chain first	6 621	9 026	75
data first	5 732	9 018	75
round robin	6 624	9 020	75
fullest first	6 665	8 993	75

VII. Conclusion

This papers presents a Live-Tracing architecture for RTL designs, which captures signals of a system continuously, transmits them to an external interface, and converts them to the VCD format compatible with most waveform viewers. The architecture allows to trace designs without reducing or stopping the clock, which makes the design feasible for debugging off-chip interfaces or real-time applications. The Live-Tracing-Logic contains the *Tracing-Controller*, which is connected to the external interface and the *Tracing-Modules*, which are connected in a chain. The modules are parameterizable and allow to trace signals of different bit-widths, clocks, and numbers of triggers. The key of the Live-Tracing-Logic is designed for debugging purposes and is integrated automatically. Similar to ILA, the Live-Tracing-Logic captures data after an event was detected by a trigger and is controlled by a host system software. Also in common to the Xilinx ILA is the parameterizable trigger logic, by in/excluding defined triggers or setting compare values during the debugging time. In contrast, we are not limited to windows of captured data, which is achieved by sending only traced data if the trigger condition is true and a value change of the signals occurs. The Live-Tracing architecture was evaluated by a test design connected via PCIe, with 8 modules, which generates data continuously. The results show that a trace of an overall bandwidth of up to 3.10 GB/s without any X-event is possible. The utilization for LUTs and FlipFlops are 67 % respectively 70 % lower compared to the ILA of Xilinx. The memories of the design can be configured for different use cases and for an expected bandwidth of up to 3.10 GB/s, only very small internal memories for the *Tracing-Modules* are required. The design also better adapts to designs with different clocks, as no more resources are required.

References

[1] S. Asaad, J. Tierno, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, and T. Takken, "A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multi-core processor simulation," in *Proceedings of the ACMSIGDA international symposium on Field Programmable Gate Arrays*, ser. ACM Digital Library, K. Compton, Ed. New York, NY: ACM, 2012, p. 153.

[2] Xilinx, "Integrated Logic Analyzer v6.2: LogiCORE IP Product Guide," 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf

[3] ——, "Vivado Design Suite User Guide: Programming and Debugging," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug908-vivado-programming-debugging.pdf

[4] IEEE, *IEC 61691-4 First edition 2004-10; IEEE 1364: IEC/IEEE Behavioural Languages - Part 4: Verilog Hardware Description Language (Adoption of IEEE Std 1364-2001)*. [S.l.]: IEEE, 2004.

[5] Altera, "Quartus II Handbook Volume 3: Verification," 2015. [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-prime/features.html

[6] J. Goeders and S. J. Wilton, "Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits - IEEE Xplore Document," 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7466842/

[7] J. P. Pinilla and S. J. E. Wilton, "Enhanced source-level instrumentation for FPGA in-system debug of High-Level Synthesis designs," in *Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT)*. [Piscataway, New Jersey]: IEEE, 2016, pp. 109–116.

[8] E. Hung and S. J. E. Wilton, "Incremental Trace-Buffer Insertion for FPGA Debug," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 4, pp. 850–863, 2014.

[9] F. Eslami and S. J. Wilton, "An adaptive virtual overlay for fast trigger insertion for FPGA debug," in *2015 International Conference on Field-Programmable Technology (FPT)*. [Piscataway, NJ] and [Piscataway, NJ]: IEEE, 2015, pp. 32–39.

[10] M. Jassi, B. Bordes, D. Muller-Gritschneder, and U. Schlichtmann, "Automation of FPGA performance monitoring and debugging Using IP-XACT and graph-grammars," in *2015 International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*. Piscataway, NJ: IEEE, 2015, pp. 1–4.

[11] A. Kourfali and D. Stroobandt, "Efficient Hardware Debugging Using Parameterized FPGA Reconfiguration," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016)*. Piscataway, NJ: IEEE, 2016, pp. 277–282.

[12] Z. Panjkov, A. Wasserbauer, T. Ostermann, and R. Hagelauer, "Automatic debug circuit for FPGA rapid prototyping," in *SISY 2015*. [Piscataway, New Jersey]: IEEE, 2015, pp. 155–160.

[13] H. ul Hasan Khan and D. Göhringer, "FPGA debugging by a device start and stop approach," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2016, pp. 1–6.

[14] Z. Panjkov, A. Wasserbauer, T. Ostermann, and R. Hagelauer, "Hybrid FPGA debug approach," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–8.

[15] E. Billauer, "Xillybus," Haifa, 2017. [Online]. Available: http://xillybus.com/