FPGA-Based Debugging with Dynamic Signal Selection at Run-Time

Gernot Fiala
Graz University of Technology
Graz, Austria
gernot.fiala@student.tugraz.at

Tobias Scheipel
Graz University of Technology
Graz, Austria
tobias.scheipel@tugraz.at

Werner Neuwirth
AVL List GmbH
Graz, Austria
Werner.Neuwirth@avl.com

Marcel Baunach
Graz University of Technology
Graz, Austria
baunach@tugraz.at

***Abstract*—For the development of FPGA-based automotive systems, debugging of internal signals is necessary to detect errors or to analyze/visualize the operation of the field programmable gate array (FPGA) at runtime. Often, so called "debug cores" of the FPGA vendor are used for debugging. Xilinx Vivado is a development environment offering an integrated logic analyzer for statically selected signals. However, each time these input signals shall be changed, the whole workflow (synthesis, placement, routing and generation of the bit stream) must be repeated, which is very time consuming.**

The scope of the present work is to develop a custom and more flexible FPGA-based logic debugger: The Advanced Inverter Debugger (AID) is a logic component, integrated into the system under development, that can dynamically select signals for the debugging process at run-time. The debugging process is controlled by a user interface at a workstation, communicating via UDP/IP over Ethernet. The AID is configurable with regard to start/stop triggers and sample rate for each signal, and allows long-term recording as well as visualization at the workstation. For convenient use in the development of automotive control systems, the AID is available as Matlab component for integration into and synthesis with the target system.

***Index Terms*—automotive, debugging, FPGA**

I. Introduction

In the automotive industry, electric engines are becoming more and more important. For testing synchronous and asynchronous engines, complex test benches are used, which are able to set up different test conditions. These test benches use FPGA-based inverters and controllers, which provide several functionalities, e.g., pulse width modulation (PWM), phase-locked loop (PLL), voltage and current control to power the different synchronous and asynchronous engines. During the development process, these inverters and controllers must continuously be checked for correct operation, which is done by debugging internal signals of the FPGA design.

For debugging, standard debug cores of the FPGA vendor like the integrated logic analyzer (ILA) [1] provided by Xilinx Vivado [2] are commonly used. The ILA core uses statically selected input signals and settings for the debugging process. If the input signals or the debugging settings need to be changed, the internal structure of the ILA core is updated in the FPGA design. Therefore, the whole workflow (synthesis, placement, routing and generation of the bit stream) must be repeated, which is very time consuming (especially for large FPGA designs) and requires to stop and restart the system. To be more flexible and not to interrupt running tests, improved debug cores are required.

The present work introduces a custom FPGA-based logic debugger, the Advanced Inverter Debugger (AID). The AID can dynamically select signals for the debugging process at run-time. The debugging process is controlled by a user interface at a workstation. The debugging parameters are sent from the workstation to the debug core on the FPGA. The communication is done with UDP/IP (user datagram protocol/internet protocol). The debug core decodes the command information from the user interface and autonomously starts the debugging process with the given configuration. The sampled signal data is then sent from the FPGA to the workstation and monitored with the user interface. Optionally, the signal data can be logged in comma-separated values (csv) files for long-term observation and delayed analysis.

This paper is organized as follows: Section II shows related work on debug cores. Section III gives an overview on different concepts to implement such debug cores. Section IV explains the structure and operation of the AID debug core. Section V shows the resource usage of the debug core on the FPGA. Section VI shows the test hardware on which the debug core was tested. Section VII shows the user interface, which controls the debug core and tests. Finally, Section VIII concludes the paper.

II. Related Work

Xilinx Vivado provides the ILA core, which allows developers to put an integrated logic analyzer into their FPGA designs. An ILA can monitor signals during the execution of the system at a predefined sampling rate if the signals meet predefined trigger conditions. The logic overhead varies depending on the selected number of samples and the defined input signals. The samples are stored on the FPGA and sent via a JTAG interface to the workstation for monitoring with Xilinx Vivado.

Debugging and validation of logic in FPGA designs was also considered by various researchers. In [3], a method of run-time debugging and monitoring of FPGAs is shown. An embedded microprocessor is used to monitor internal signals

of the FPGA design. The connection between the signals and the microprocessor works via the on-chip memory (OCM) bus and the processor's local bus (PLB) through the shared memory.

Another method uses a scan-chain based approach [4]. A watch-point capability is provided by inserting a scan-chain into the FPGA design, which is configured as shift register. With this method it is possible to change the watch-point conditions at run-time without a recompilation of the FPGA design. However, the monitored signals can not be changed at run-time.

III. Communication concepts for the debug core

The debug core was initially designed to debug internal signals of inverters and voltage and current controller for automotive test benches. To debug these internal signals and to visualize the inverter and controller behavior, the AID provides an interface for up to 300 possible input signals. Out of these, 4 can be selected dynamically for the debugging process at run-time. The 300 input signals were chosen to have a large selection of the internal signals of the controller. Due to the high number of signals, big multiplexers are required. To lower the resource usage on the FPGA, only 4 signals can be selected concurrently for the debugging process. The debug core sends the sampled signal data continuously to the user interface, depending on the sampling frequency. Once started, this allows infinite debugging processes.

For the inclusion of the debug core into the FPGA design, 3 different methods are discussed in this section.

A. Communication with UDP/IP and the ARM Processor

With this approach, the communication between the debug core and the user interface at the workstation is done via UDP/IP and the processing system (PS) [5] of an ARM processor, shown in Fig. 1. The Ethernet interface of the ARM processor is used for the communication with the user interface. An UDP echo server is running as standalone application on the ARM processor, which is responsible for receiving and sending the UDP packages. This application uses the Lightweight Internet Protocol (lwIP) library, which is a network stack for embedded systems.

To connect the programmable logic (PL) with the PS of the ARM processor, the Advanced eXtensible Interface (AXI) [6], a general purpose input output (GPIO) [7] port, an interrupt system (enabled interrupt controller), and the direct memory access (DMA) are used. The processor RAM is used for transferring data between the PS and the PL. The AXI GPIO port enables the read operation from the RAM on the PL side. The AXI-DataMover [8] reads the debugging parameters from the RAM and routes them to the DebugCore block.

The connection from the PL to the PS is done with interrupts and DMA. The sampled signal data is written via AXI-DataMover into the RAM and the interrupt is set. The PS processes the corresponding interrupt and reads the signal data from the RAM, builds the UDP package and sends it to the workstation with the user interface.

The lwIP library also allows to use the transmission control protocol (TCP) for the connection between the ARM processor and the workstation. However, due to lower communication overhead with UDP/IP, smaller data packages can be sent faster from the ARM processor to the workstation. Therefore, UDP/IP was selected.

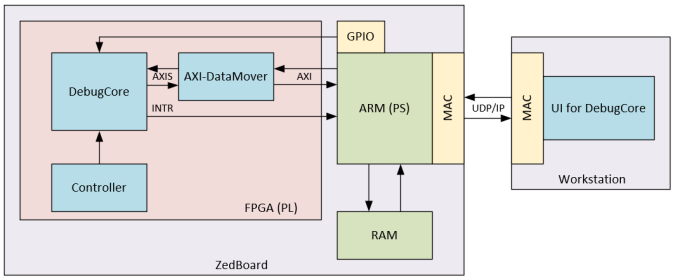

Fig. 1. Communication between the debug core and the user interface with UDP/IP and the ARM processor.

B. Communication with UDP/IP and the AXI-Ethernet IP core

This approach uses the AXI-Ethernet [9] IP core for the communication between the debug core and the user interface, shown in Fig. 2. The debugging parameters are adjusted with the user interface and sent with UDP/IP to the media access control (MAC) interface. The AXI-Ethernet IP core from Xilinx is used to receive the data and route it via an AXI-Stream (AXIS) [6] bus to the DebugCore block. The sampled signal data can be stored in registers to collect enough samples for a UDP package. Then the collected signal samples are sent via the AXIS bus to the AXI-Ethernet block, which converts the AXIS data into data for the Ethernet transceiver and builds and sends the UDP package to the workstation. The user interface processes and monitors the signal data.

With this approach no PS and ARM processor is required. There is also no need for the DMA to access the RAM because the signal samples can be collected for a UDP package in registers on the FPGA. This might slightly increase the resource usage on the FPGA but can be accepted.

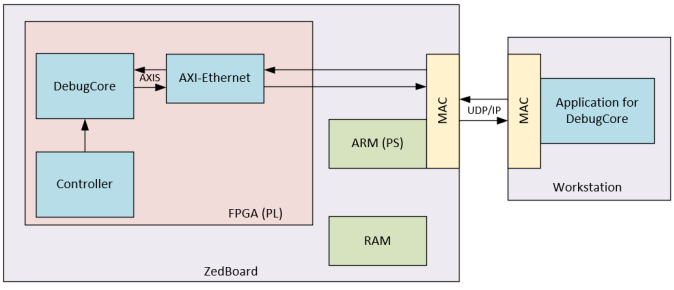

Fig. 2. Communication between the debug core and the user interface with UDP/IP and the AXI-Ethernet IP core.

C. Communication with TCP/IP and the AXI-Ethernet IP core

This approach also uses the AXI-Ethernet IP core for the communication between the debug core and the user interface, shown in Fig. 3. The debugging parameters are sent with TCP/IP from the user interface to the MAC interface. The AXI-Ethernet IP core receives the data and routes it via AXIS bus to the DebugCore block. TCP/IP allows bigger data packages and more signal data can be sent to the user interface. Therefore, sampled signal data can be stored in the RAM for the TCP package payload. To write the signal data into the RAM, the AXI-DataMover is used. When enough samples are collected, the data is read from the RAM and transferred to the AXI-Ethernet IP core. The Ethernet transceiver sends the TCP package to the user interface at the workstation.

With this approach, no PS and ARM processor is required. The RAM can be accessed with the AXI-DataMover and DMA to collect the signal samples for the TCP package. This requires a unit, which controls the write and read operations via the AXI-DataMover. This concept is more complex compared to the others, because memory access and the inclusion of the AXI-Ethernet IP core is required.

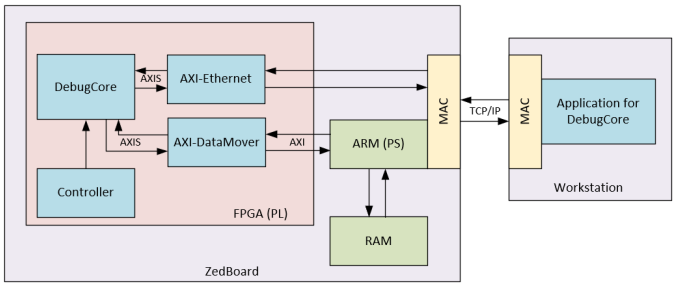

Fig. 3. Communication between the debug core and the user interface with TCP/IP and the AXI-Ethernet IP core.

To use the AXI-Ethernet IP core, a licence for the Xilinx Tri-Mode Ethernet Media Access Control (TEMAC) [10] is necessary.

D. Decision for the Implementation

The idea was to send the sampled signal data as fast as possible to the workstation to minimize the used memory resources on the FPGA. With UDP/IP, packages can be sent faster compared to TCP/IP and potential package loss with UDP was not an issue during our evaluations and in-field tests. Both concepts with the AXI-Ethernet IP core are processor independent, which is a big advantage for the integration of the AID into different FPGA designs. Unfortunately, due to the missing TEMAC licence during the development time, the AXI-Ethernet IP core couldn't be used and both concepts with the AXI-Ethernet IP core were not possible for the implementation. To evaluate the functionality of the AID and to analyze the communication with UDP/IP, the decision was made to implement the first concept, "Communication with UDP/IP and the ARM processor".

IV. Design and Implementation of the Debug Core

A. Structure of the Debug Core

The debug core was developed in VHDL (very high speed integrated circuit hardware description language). It is built with different units shown in Fig. 4. The MM2S-Datamover interface is the AXIS interface for the Memory Mapped to Stream (MM2S) transfer. The interface is used, when the AXI-DataMover reads the debug parameters from the RAM and transfers it to the debug core. With the unpackage (UNPKG) Module, the AXIS data is decoded into the different parameters to set up the debugging process. The decoded signals are routed to the Debug Core Module, which is the heart of the debug core. It is responsible for the trigger, signal selection and signal sampling. The Debug Core Module unit was developed with Matlab [11] SIMULINK [12]. The VHDL code was generated with the Matlab model and included into the Vivado project. The sampled signal values are transferred to the package (PKG) Samples unit, which builds the AXIS data stream. The AXIS data stream is transferred via the Stream to Memory Mapped (S2MM) Datamover interface to the AXI-DataMover, which writes the data into the RAM. The Datamove control (CTL) unit controls the AXI-DataMover operations. It sets an interrupt, when the debug parameters are successfully read from the RAM and transferred to the debug core. It also signals the PS with an interrupt when signal data can be read from the RAM to build the UDP package and send it to the workstation.

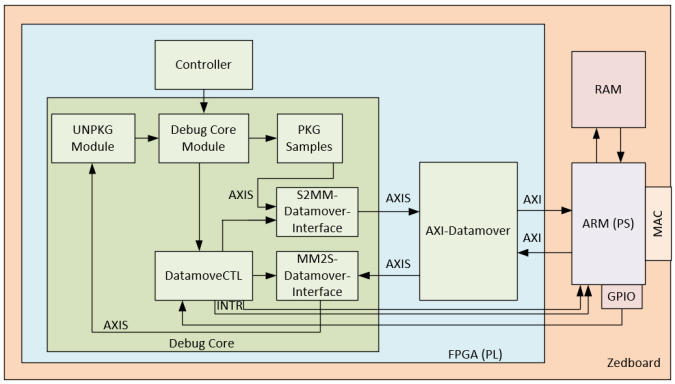

Fig. 4. Debug core structure on the FPGA

B. Implementation of the debug core

The debug core has up to 300 possible input signals, each 32 bits wide. They are combined to a signal interface and 4 signals of them can be selected for the debugging process.

The signal selection is part of the Debug Core Module block. To select the 4 signals, 4 big multiplexers are used. In Matlab SIMULINK, a 300:1 multiplexer can be built very easily, but when the VHDL code is generated and synthesized, the multiplexers are built with the available F7 and F8 multiplexers of the FPGA. Since these multiplexers are commonly needed for the main functions of the controllers and inverters, our

300:1 multiplexers are designed to us Look-up Tables (LUT). This was done by using bit wise logical disjunction.

To select the signals for the debugging process, the control information from the user interface is decoded and routed as selection signals into the signal selection blocks. Depending on the value, one of the 300 input signals is selected as output signal. The structure of the signal selection is shown in Fig. 5. The signal selection can be done at run-time.

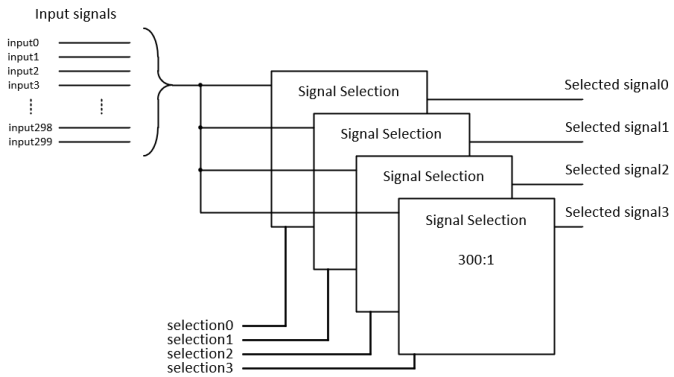

Fig. 5. Structure of the signal selection

The selected signals can be sampled with the adjusted sample frequency. The operation frequency is 1 MHz for debug core but all modules using the AXIS interface are operating at a frequency of 100 MHz. The maximum sample frequency is limited to 1 MHz and the lowest sample frequency is 1 kHz. Overall, there are 25 different sampling frequencies for the debugging process available, which can be selected in the user interface. An internal counter controls the sample points depending on the adjusted sample frequency. The functionality is comparable to a frequency divider.

The number of samples defines how long the debugging process is active. The selectable values are between 1024 and 999424 samples. The step size is 1024. An internal counter increases with each sample and when the adjusted number of samples is reached, an internal reset occurs, which stops the debugging process and resets all of the debug core sub modules. This operation mode is called normal mode. There is a second operation mode, the infinity mode, in which the sampling process is active until the stop command information is sent from the user interface to the FPGA.

To influence the start condition, a trigger can be enabled. There is a pre- and a post-trigger available. The post-trigger activates the sampling process immediately when the trigger condition is met and the sampled signal data is sent to the user interface. The pre-trigger continuously saves 100 samples per signal into a ring buffer. Once the trigger condition is met, the signal data is sent from the ring buffer to the user interface. This allows to also analyse the signals before the actual trigger event.

To access the Block RAMs (BRAM) of the ring buffer, counters are used as write and read addresses. When the start debugging command was received, the write address increases with each sample point. If no trigger event happens and 100 signal values are written into the BRAMs, the write address overflows and starts at 0 to overwrite the old signal values. The read address starts to increase and follows the write address with a constant gap. The read address also increases, when the trigger condition is met. Then, the values of the ring buffer are used to build the UDP packages. This gives information about the signal behaviour before the trigger event occurs.

Trigger condition can be set for the post- and pre-trigger. The available trigger conditions are above, lower or equal to the trigger value, shown in Table I. The trigger signal is always the first selected signal. When no trigger is active, the sampling process starts directly after sending the start debugging command information from the user interface to the FPGA.

TABLE I
TRIGGER TYPES OF THE DEBUG CORE

Trigger Type	Trigger condition
above value	signal value is above trigger value
lower value	signal value is lower than trigger value
equal to	signal value is equal to trigger value

C. Communication between the Programmable Logic and the Processing System of the ARM processor

The communication from the PS to the debug core on the FPGA is done with AXI-GPIO ports, shown in Fig. 4. They support a configurable I/O channel width of up to 32 bits. These AXI-GPIO ports can be addressed with driver files from the PS. AXI-GPIO ports are used to initialize the read operation from the RAM with the AXI-DataMover. To configure the transfers between the AXI-DataMover and the RAM also AXI-GPIO ports are used.

The communication from the debug core to the PS is done with interrupts. The first interrupt signals the PS that the debug parameters were successfully read from the RAM and the AXI-GPIO port for initializing the read operation can be reset.

The second interrupt signals the PS, that the sampled signal values were successfully written into the RAM and can be read with the PS, to build the UDP package and send it to the workstation. To prevent simultaneous memory access, 2 memory addresses are used alternately to write the sampled signal data into the RAM. The DataMoveCTL block controls the alternating address change. Multiple samples can be collected for the UDP transfer, before the interrupt is set. The adjustments for this can be made in the IP settings of the DataMoveCTL block or with the PS and AXI-GPIO ports. Currently 32 samples are used to build the UDP package. One sample contains the package type, sample number (timestamp) and the 4 signal values.

D. Communication between the user interface and the ARM processor

The communication between the user interface and the ARM processor is done with UDP/IP. Different package types are defined, to distinguish between control information,

version number and signal data. The package type defines, which information are transmitted with the UDP package. The different package types are shown in Table II. A standalone application with a UDP echo server is running on the ARM processor. It receives and sends the UDP packages.

If a UDP package is received and the command data are written into the RAM, the AXI GPIO port to enable the read operation is set. The corresponding interrupt is processed by the interrupt system and the program waits for the interrupts to read the signal data from the RAM to build the UDP package and send it to the workstation.

The version number request is directly processed by the PS, which sends the version number back to the user interface.

TABLE II
DEFINITION OF THE PACKAGE TYPES FOR THE UDP CONNECTION

Package Type	Description
0	command information to start the debugging process
1	signal data
3	command information to reset the debugging process
5	request for the version number
6	version number acknowledgement

V. RESOURCE USAGE OF THE DEBUG CORE ON THE FPGA

The signal selection logic is the biggest part of the debug core. The multiplexers with 300 input signals, each 32 bits, need a significant amount of resources on the FPGA in order to allow selection flexibility. To lower the resource usage, a debug core with 40 possible input signals was created. A comparison between the AID300 and AID40 is shown in Table III. The FPGA on the ZedBoard [13] was used to get an overview of the used resources. The biggest differences between the AID40 and AID300 are visible at the ”Slice LUT”, ”Slice” and ”LUT as Logic” counters. This differences are caused by the reduction of the input signals. However, no F7 and F8 multiplexers are required, because the signal selection logic was developed to avoid them.

TABLE III
RESOURCE USAGE OF THE AID40 AND AID300 ON THE ZEDBOARD

Resource	ZedBoard	AID40	AID300
Slice LUT	53200	2055	5616
Slice Register	106400	1420	2500
Slice	13300	755	1730
LUT as Logic	53200	2055	5616
LUT Flip Flop Pairs	53200	257	257
Brock RAM Tile	140	2	2
DSP	220	1	1

VI. TEST-HARDWARE

The debug core was tested with the ZedBoard [13]. The ZedBoard is a development board for the Xilinx Zynq-7000 System on Chip (SoC) [14]. It contains a dual-core ARM Cortex-A9 processor and a Z-7020 FPGA [15]. Several interfaces like, e.g., Universal Asynchronous Receiver Transmitter (UART), Universal Serial Bus (USB), JTAG, High Definition Multimedia Interface (HDMI), Video Graphics Array (VGA), Audio I/O, Ethernet are supported and can be used for different kinds of applications. It also includes Double Data Rate Random-Access Memory (DDR3-RAM), an interface for Secure Digital (SD) memory card, Light-Emitting Diodes (LEDs), switches and I/O interfaces. With the processing system the different components can be activated and the programmable logic of the FPGA can be configured.

VII. USER INTERFACE AND TESTS

The user interface controls the debug core on the FPGA. All adjustments for the UDP connection and the debugging process can be made here. The UDP connection is set up with the IP address, incoming and outgoing port numbers.

To get the signal names from the FPGA design, which are connected to the debug core, a signal configuration file generator was implemented with C#. This file generator extracts the input signal names with the interface name of the debug core of the FPGA design and maps them to the input port numbers to generate a csv configuration file. This configuration file can be loaded into the user interface to display the signal names, which can be selected for the debugging process.

The user interface was programmed with C# and tested with the ZedBoard. Fig. 6 shows the user interface with the different settings for the debugging process. The debugging process was started with 8192 samples and a sample frequency of 1 MHz. The post-trigger is selected as trigger condition and data logging is enabled, which generates a csv file and saves the signal data with the debugging settings.

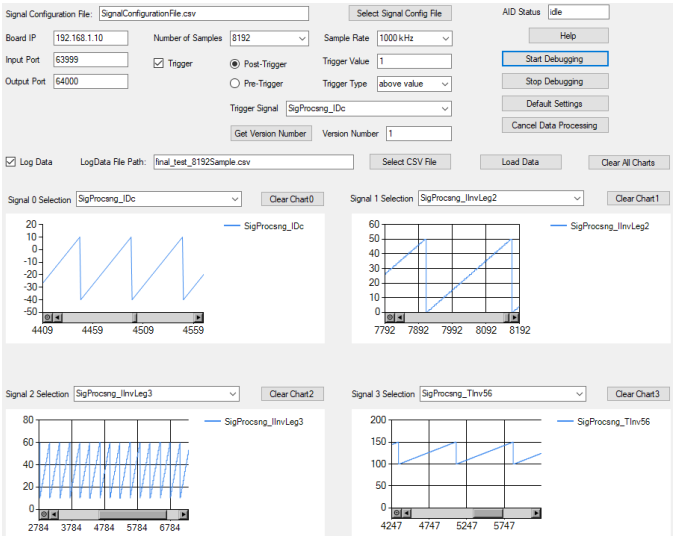

Fig. 6. Testing the debug core with the ZedBoard and the signal generator

After each received 999424 samples, a new file is automatically created. This is important, when the debugging process is running in infinity mode. The csv files with the

logged signal data can also be loaded and monitored with the user interface for delayed analysis.

To test the debug core, a signal generator is used to simulate a complex FPGA-based automotive system. It generates 300 test signals, which are used as input signals for the debug core. The first 20 test signals are simple counters, which start at different values and increase with different frequencies. The other test signals are constants to test the signal selection of the debug core. The test setup is a hardware in the loop test by running the FPGA design on the FPGA of the ZedBoard. The debugging process is started with the adjusted debugging settings of the user interface. A receiving thread and a processing thread are used to receive the UDP packages and to process, log and monitor the signal data. It works well for lower sample frequencies and high sample frequencies with lower sample numbers shown in Fig. 7. The sample frequency is adjusted to 1 MHz and the number of samples is set to 60416. However due to UDP/IP, small packages can be sent very fast from the FPGA to the workstation and when a large number of samples and a high sample frequency (1 MHz or 500 kHz) are adjusted, sample loss occurs. During the tests, the incoming UDP packages were analyzed with Wireshark [16]. Each UDP package arrived successfully at the workstation but the receiving thread gets blocked sometimes by other threads and can not process the incoming UDP packages fast enough and the samples are lost. This is shown in Fig. 8 with the adjusted sample frequency of 1 MHz and 100352 samples. UDP packages can also arrive in the wrong order because UDP does not support packet sequencing. Therefore, the packages are numbered by the AID upon transmission.

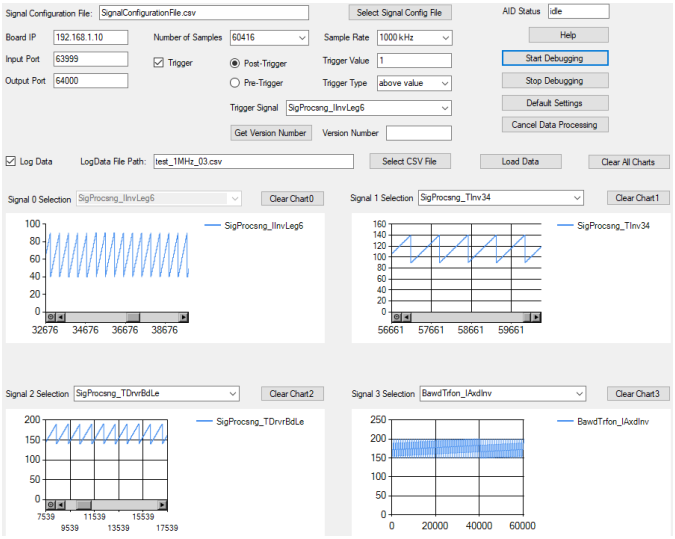

Fig. 7. Testing the debug core with the ZedBoard with 1 MHz sampling rate and 60416 samples

VIII. Conclusion

This paper shows a custom FPGA-based debug core, the Advanced Inverter Debugger (AID), which was initially developed to debug FPGA-based inverters and controllers. The AID is able to dynamically select signals for debugging FPGA-based embedded automotive systems at run-time. In order to be flexible with the signal selection, the possible number of input signals has to be large. However, the multiplexers for the signal selection requires more resources on the FPGA, which might not be available.

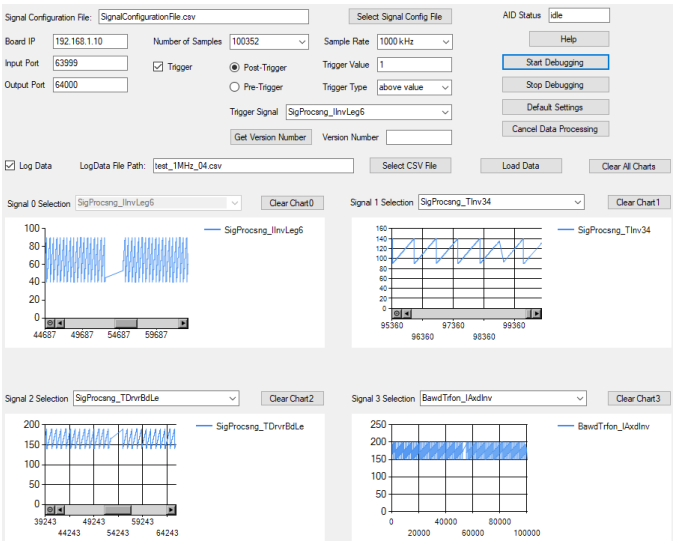

Fig. 8. Testing the debug core with the ZedBoard with 1 MHz sampling rate and 100352 samples

The AID is controlled by a user interface at a workstation. The communication is done with UDP/IP and the processing system (PS) of the ARM processor. To transfer the data between the programmable logic (PL) and the PS, the processor RAM is used.

The limitations of this approach are the receiver of the workstation and the PS of the ARM processor. With UDP/IP, packages can be sent very fast but the user interface at the workstation has problems to process the data at high sample frequencies. The receiving thread of the user interface is blocked sometimes, which leads to sample loss. There are also limitations when the PS is used for other operations beside the AID. Some interrupts of the debug core might no longer be processed, UDP packages are not sent, and samples get lost.

For the future, the TEMAC license can be acquired to use the AXI-Ethernet IP core. With this IP core, the communication can be done without the PS but the flexibility of the AID remains the same. Also, the communication can be changed to TCP/IP. With TCP/IP, bigger packages can be sent and the receiver has more time to process the received data.

Acknowledgment

This research was supported by AVL List GmbH, Graz, Austria.

References

[1] Xilinx, “Integrated logic analyzer v6.1,” Xilinx, Apr. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf

[2] ——, "Xilinx vivado," Xilinx. [Online]. Available: https://www.xilinx.com/products/designtools/vivado.html
[3] A. Penttinen, R. Jastrzebski, R. Pöllänen, and O. Pyrhönen, "Run-time debugging and monitoring of fpga circuits using embedded microprocessor," in *2006 IEEE Design and Diagnostics of Electronic Circuits and systems*. Prague, Czech Republic: IEEE, Apr. 2006, 1-4244-0185-2.
[4] A. Tiwari and K. A. Tomko, "Scan-chain based watch-points for efficient run-time debugging and verification of fpga designs," in *2003 Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference*. Kitakyushu, Japan: IEEE, Jan. 2003, 0-7803-7659-5.
[5] Xilinx, "Processing system 7 v5.5," Xilinx, May 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_5/pg082-processing-system7.pdf
[6] ——, "Axi reference guide," Xilinx, July 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf
[7] ——, "Axi gpio v2.0," Xilinx, Oct. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf
[8] ——, "Axi datamover v5.1," Xilinx, Apr. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v5_1/pg022_axi_datamover.pdf
[9] ——, "Axi 1g/2.5g ethernet subsystem v7.0," Xilinx, Apr. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_0/pg138-axi-ethernet.pdf
[10] ——, "Tri-mode ethernet mac v9.0," Xilinx, Apr. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/tri_mode_ethernet_mac/v9_0/pg051-tri-mode-eth-mac.pdf
[11] MathWorks, "Matlab." [Online]. Available: https://de.mathworks.com/products/matlab.html
[12] ——, "Simulink." [Online]. Available: https://de.mathworks.com/products/simulink.html
[13] AVNET, "Zedboard." [Online]. Available: http://zedboard.org/product/zedboard
[14] Xilinx, "Zynq-7000 soc data sheet: Overview," Xilinx, July. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
[15] ——, "Zynq-7000 soc z-7020 data sheet," Xilinx, July. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf
[16] Wireshark-Community, "Wireshark." [Online]. Available: https://www.wireshark.org/