Gernot Fiala, BSc

AID – Advanced Inverter Debugger
Reconfigurable FPGA based Debugger

MASTERARBEIT

Zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Information and Computer Engineering

Eingereicht an der

Technischen Universität Graz

Betreuer

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach

Institute of Technical Informatics

Graz, December 2019

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

_____ _____

Datum Unterschrift

Graz, März 2019

ACKNOWLEDGMENTS

I want to thank all of the involved people, who motivated and supported me during my work on this master thesis.

First, I want to thank my supervisor Univ.-Prof. Dipl.-Inf. Univ. Dr. rer. nat. Marcel Carsten Baunach, who is professor at the Technical University of Graz and part of the Institute for Technical Informatics and Head of the working group Embedded Automotive Systems. With his contacts to AVL List GmbH, it was possible to offer this master thesis to me. This interesting project helped me to increase my FPGA and SoC experience.

I want to thank AVL List GmbH and particularly DI Dr. Michael Wiesinger, Head of the MH Department, DI Werner Neuwirth, Head of the MHE Department, who provided me all the resources I needed, which were necessary for this thesis and for the contract to support me financially.

I also want to thank my colleges at AVL List GmbH, Florian Mittlinger, MSc. Stephan Hochmüller, MSc, and DI Dr. Oliver König, who helped me when I had difficulties.

My deepest gratitude also goes to my family, who supported me mentally and motivated me through all the courses. A big thanks goes to my parents for always caring about me and for their financial support to enable this education.

ABSTRACT

This master thesis is about the implementation of a reconfigurable FPGA based Logic-Debugger, the AID – Advanced Inverter Debugger and was provided by the Technical University of Graz and accomplished in cooperation with AVL List GmbH. For FPGA based development, debugging of internal signals is necessary to detect errors and to visualize signals. Xilinx Vivado already offers an Integrated Logic Analyzer (ILA) to debug the signals of the design. Every time the input signals are changed, the whole workflow (synthesis, placement, routing and the generation of the bit stream) must be done again. This costs time and resources.

Therefore, it is better to use a custom Debug-Core, like the Advanced Inverter Debugger. The AID has 300 possible signal inputs. It can dynamically select 4 signals out of them for the debugging process. The debugging process is controlled by a user-interface at a workstation. The adjusted debugging parameters, like sample rate, number of samples, trigger settings and signal selection, are sent from the workstation to the Debug-Core on the FPGA. The communication is done with UDP/IP. The Debug-Core starts the debugging process with the adjusted parameters. The sampled signal data is sent from the FPGA to the workstation and monitored with the user-interface. Optional, the signal data can be logged in csv files.

The functionality of the AID was tested with the Zynq Development Board and a custom Controller Board developed by FH Kapfenberg and AVL List GmbH. Different kinds of inverters and controllers are running on the Controller Board, which are needed for test benches in the automotive industry.

In the future the AID should be used for FPGA based development and for error analysis for different kinds of inverters.

ABSTRACT

Diese Masterarbeit beschäftigt sich mit der Implementierung eines FPGA basierten Debuggers, dem AID – Advanced Inverter Debugger und wurde von der Technischen Universität Graz angeboten und in Kooperation mit der AVL List GmbH durchgeführt. Für die Entwicklung mit FPGAs ist es wesentlich, die internen Signale zu betrachten um Fehler zu erkennen und Signale darstellen zu können. Xilinx Vivado verfügt über einen Integrated Logic Analyzer (ILA), um Signale aus dem Design aufzuzeichnen. Allerdings muss jedes Mal, wenn sich die Eingangssignale vom ILA-Core ändern, der gesamte Arbeitsprozess (Synthese, Platzierung, Verkabelung und die Generierung des Bitstreams) neu durchlaufen werden, was in Folge viel Zeit und Ressourcen in Anspruch nimmt.

Um dies zu vermeiden, ist es sinnvoller einen eigenen Debug-Core zu entwickeln, den Advanced Inverter Debugger. Der AID besitzt 300 mögliche Eingangssignale. Aus diesen können jeweils 4 Signale dynamisch ausgewählt und aufgezeichnet werden. Auf einem Computer kann der Debugging-Prozess mit Hilfe einer Benutzeroberfläche gesteuert werden. Die Parameter Abtastrate, Anzahl der Abtastpunkte, Triggereinstellung und Signalauswahl werden für die Aufzeichnung in der Benutzeroberfläche eingestellt und an den FPGA gesendet. Die Kommunikation erfolgt mit UDP/IP. Der Debug-Core verarbeitet die eingestellten Parameter und startet den Debugging-Prozess. Die abgetasteten Signalwerte werden vom FPGA zurück an den Computer gesendet und mit der Benutzeroberfläche grafisch dargestellt. Die Signalwerte können optional in csv Dateien gespeichert und wieder angezeigt werden.

Die Funktionalität wurde mit dem Zynq Development Board und einem, von der FH Kapfenberg und AVL List GmbH entwickelten, Controller Board getestet. Auf diesem Controller Board laufen diverse Umrichter, die für Testeinrichtungen in der Automobilindustrie verwendet werden.

In der Zukunft soll der AID für die Entwicklung mit FPGAs und zur Fehlererkennung von verschiedenen Umrichtern eingesetzt werden.

INDEX

1 INTRODUCTION

This master thesis is about the implementation of an FPGA based Logic-Debugger in cooperation with AVL List GmbH[1], which can monitor internals signals of the FPGA design.

AVL List GmbH is a company in the automotive industry. Their main focus is the development of drive systems like engines, powertrains, batteries and the associated software, test benches for engines, vehicles and their components and simulations for engines and vehicle development.

For the development of FPGA based products, debugging of internal signals is necessary to detect errors or to visualize signals of the FPGA design. Xilinx Vivado[2] already offers an Integrated Logic Analyzer (ILA) to debug signals of the design. Every time, the input signals of the ILA-Core change, the whole workflow (synthesis, placement, routing and generation of the bit stream) must be done again. This costs time and resources.

To avoid this waste of resources, it is better to use a custom Debug-Core, like the AID – Advanced Inverter Debugger. The AID has 300 possible signal inputs. It can dynamically select 4 signals out of them for the debugging process. The debugging process is controlled by a user-interface at a workstation. The adjusted debugging parameters are sent from the workstation to the Debug-Core on the FPGA. The communication is done with UDP/IP. The Debug-Core starts the debugging process with the adjusted parameters. The sampled signal data is sent from the FPGA to the workstation and monitored with the user-interface. Optional, the signal data can be logged in csv files.

The AID should be used for debugging different kinds of inverters in the automotive industry. These inverters are used to convert voltages and currents from AC to three-phase and vice versa. They also provide several functionalities like PWM, PLL, voltage and current control etc. for test benches to power synchronous and asynchronous engines.

First, there is a little overview about different concepts to implement this Debug-Core. These concepts have different advantages and disadvantages. A comparison was made, which concept is better and also feasible with the existing tools.

Second, there is an overview of the whole system and how the different parts interact with each other. The main parts are the Debug-Core on the FPGA and the user-interface for the workstation.

After the short overview, the different parts of the Debug-Core and the user-interface are explained in detail and how they were implemented. These parts were simulated and tested to check if they work properly.

After the simulations of the different parts, the whole system was combined and tested on the Zynq Development Board[3] and on the Controller Board. The Controller Board is a custom PCB for test benches and was developed by FH Kapfenberg[4] and AVL List GmbH.

---

[1] AVL List GmbH is a company in the automotive industry, which develops drive systems, simulations and test benches, www.avl.com
[2] Xilinx Vivado is a development tool for FPGA based development, https://www.xilinx.com/products/design-tools/vivado.html
[3] The ZedBoard is a development kit, http://zedboard.org/product/zedboard
[4] FH Kapfenberg is a college, www.fh-joanneum.at/hochschule/standorte/kapfenberg/

2 MOTIVATION

The motivation behind this thesis is to develop an FPGA based Logic-Debugger, which can debug internal signals of the FPGA design during runtime. A user-interface controls the Debug-Core and displays the sampled signal data on a workstation.

AVL List GmbH is a company in the automotive industry and develops drive systems like engines, powertrains, batteries and the associated software, test benches for engines, vehicles and their components and simulations for engines and vehicle development.

For some products, like test benches, FPGA based inverters and controllers are required. These inverters and controllers provide several functionalities like PWM, PLL, voltage and current control etc. to power synchronous and asynchronous engines. Currently, Zynq-7000 Kintex®-7 FPGAs[5] are used for the development. To verify the correct work of the inverters and controllers, debugging is necessary.

Currently, the Xilinx Integrated Logic Analyzer (ILA) is used to debug internal signals of the FPGA design. Every time, the input signals of the ILA-Core change, the whole workflow (synthesis, placement, routing and generation of the bit stream) must be done again. This costs time and resources.

To avoid this waste of resources it is better to use a custom Debug-Core, the Advanced Inverter Debugger (AID). The AID has 300 possible signal inputs. 4 of these 300 signals can be dynamically selected for the debugging process. The debugging process is controlled by a user-interface on a workstation. The adjusted parameters like sample rate, number of samples, trigger settings and signal selection are sent to the Debug-Core on the FPGA. The communication is done with UDP/IP. The Debug-Core starts the debugging process with the adjusted parameters and sends the sampled signal data back to the workstation. The signal data is displayed with the user-interface and optional logged into csv files. The logged data can also be displayed with the user-interface.

[5] Zynq-7000 Kintex®-7, https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

2.1 GOALS

To control the Debug-Core, a LabVIEW[6] user-interface should be realized. The communication between the LabVIEW user-interface and the FPGA should be done with UDP/IP. During the development this goal was changed to a C# user-interface, because C# is more flexible with file operations than LabVIEW.

The requirements for the Debug-Core are:

- Selection of 4 signals out of 300 input signals. The AID should contain 300 possible input signals, each 32 bit. 4 signals of these 300 signals should be selectable for the debugging process. To observe the behavior of the inverters and controllers, internal signals of the FPGA design must be connected to the AID IP-Core. The 300 input signals are necessary to be flexible in the selection of the signals for the debugging process.
- Adjustable number of sample points. The number of samples applies to all 4 selected signals and the AID should stop the debugging process when the adjusted number of samples is reached. The sample number is between 1024 and 999424. There should be the possibility to sample longer, but in this case, the debugging process must be stopped with the user-interface.
- Adjustable sampling rate. The sampling rate applies to all 4 selected signals and determines, how fast the sampling is done. The lowest sample frequency should be 1 kHz and the maximum sample frequency should be 1 MHz. The maximum sample frequency is determined by the operating frequency of the different controllers.
- There should be different trigger types, like post- and pre-triggers with the functionality to check, if the signal value is above, lower, or equal to the adjusted trigger value of the user-interface. The trigger option should be available for only one signal.
- Adjustable trigger value. The trigger value should be set in the user-interface. It should be an integer value, because most of the internal signals of the controllers are integer signals.
- There should be different package types for the UDP/IP connection. The package types determine how to handle the data from the UDP packages. To control the Debug-Core with the user-interface, configuration data is used with the package types to start and stop the debugging process and to get the AID version number. The Debug-Core sends data packages to the user-interface with the sampled signal data or the AID version number.
- The resource usage on the FPGA should be as low as possible.
- The controller must work independently from the Debug-Core.
- The Debug-Core is not allowed to influence the controller.
- The Debug-Core should be controlled from a workstation with a user-interface.
- The user-interface should monitor the signal data. Optional, data logging in csv files should be possible.

The implementation should be done with MATLAB[7] Simulink[8] and later with Xilinx Vivado. The user-interface should be designed with NI LabVIEW. An analysis for the data transmission should be made and the internal routing of the AID should be analyzed.

[6] LabVIEW is a graphical development tool, https://www.ni.com/de-at/shop/labview.html
[7] MATLAB is a development tool, https://de.mathworks.com/products/matlab.html
[8] Simulink is a toolbox for MATLAB, https://de.mathworks.com/products/simulink.html

3 CONCEPT OF THE DEBUG-CORE

Currently for debugging signals in the FPGA design, the Xilinx Integrated Logic Analyzer (ILA) is used. This works fine, but every time when new signals are routed into the ILA core, the whole design flow, with synthesis, placement, routing and generation of the bit stream has to be done again. Furthermore, the ILA core needs a number of sample points for the debugging process. Every time the number of sample points change, the work flow must be done again. Due to the slow data transmission with JTAG, the signal data is stored on the FPGA and sent via the JTAG interface to the workstation. To avoid a repetition of the design flow, when changes are made, more signals are into the ILA core, although some of these signals are never used for debugging. This also leads to more resource usage on the FPGA to save the signal data for the JTAG transmission.

Figure 1 shows the current status how a controller is debugged by using the ILA core and the JTAG interface. Here, internal signals of a controller are debugged to verify the correct work. The controller will be used for test benches in the automotive industry. The JTAG interface is connected to a workstation with Xilinx Vivado to monitor the debugged signals. The Controller is set up with parameters from another workstation with a LabVIEW user-interface. The communication between the Controller IP core and the LabVIEW user-interface is done with TCP/IP and with the ARM-Processor of the ZedBoard. The ARM-Processor receives the incoming TCP/IP packages and forwards them with the AXI4-Lite bus to the Controller IP core on the FPGA. The Controller sends the controller information back to the ARM-Processor, which builds a TCP/IP package and sends it to the workstation with the user-interface.

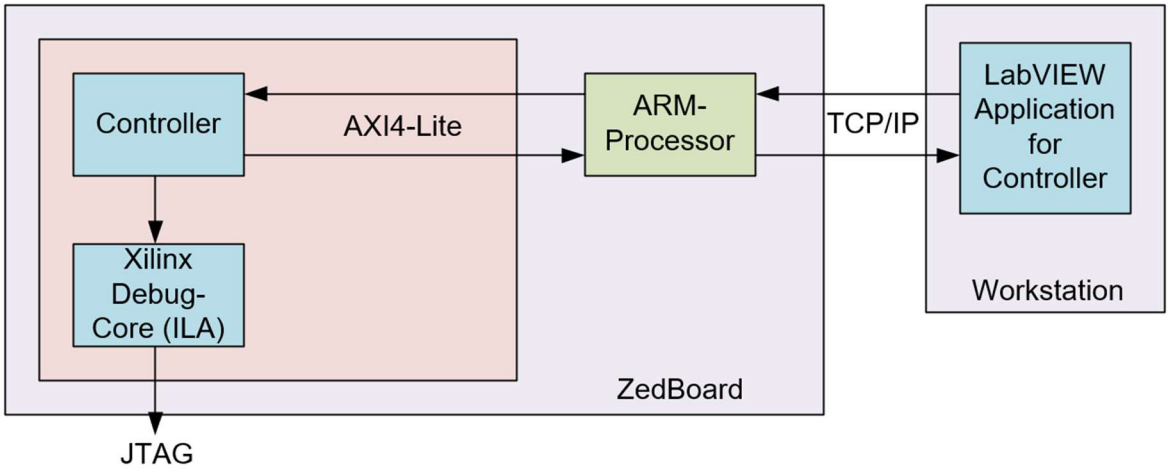


FIGURE 1: XILINX ILA CORE WITH JTAG COMMUNICATION

The ZedBoard is used for the FPGA based development. It is a complete development kit with a Xilinx Zynq®-7000 All Programmable SoC. Several interfaces, like UART, JTAG, HDMI, VGA, Audio I/O, Ethernet etc. are supported and can be used for different kinds of applications. The Zynq®-7000 SoC is structured into the Processor Subsystem, which contains the clock, reset, DDR3 RAM and Multiplexed I/O (MIO), like SD Card, Gb Ethernet, USB UART, LED, switches and other I/O interfaces. The second part is the Programmable Logic (PL), which can be configured. The PL can access interfaces like, XADC, clock, LEDs, switches, JTAG etc.[9]. The ZedBoard also includes a Xilinx Vivado® Design Edition license voucher, which is locked to Zynq-7Z020.

[9] ZedBoard, http://zedboard.org/product/zedboard

Currently, a controller for voltage and current control, which was develop by AVL List GmbH, is running on the FPGA of the ZedBoard. This controller is controlled by a LabVIEW user application. The communication is done with TCP/IP and with the ARM-Processor. A TCP/IP echo server is running on the Processing System (PS) [1], which handles the communication by using the integrated Ethernet interface of the ARM-Processor.

The ARM-Processor is the heart of the Processing System, which also includes on-chip memory, external memory interfaces and a lot of I/O peripherals. The PS offers the whole functionality of the ARM-Processer. It is possible to enable different functionalities with driver files[10]. These driver files, like a TCP/IP echo server, are executed on the ARM-Processer.

The PS receives the configuration data from the user-interface and forwards it via AXI-Lite [2] bus to the Controller IP core. The Controller sends the status data back to the workstation.

The concept of the Advanced Inverter Debugger (AID) is, to implement an FPGA based Debug-Core, which can select signals for the debugging process during runtime. This Debug-Core should be able to select 4 signals out of 300 possible input signals. To control the debugging process, a user-interface is necessary. All of the important control information can be set in the user-interface. Important parameters are the number of samples, sampling rate, trigger type, trigger value and the selection of the 4 signals for the debugging process. The parameters will be sent as control information to the FPGA. The Debug-Core will set up the debugging process with the control information and sends the signal data to the workstation. The communication between the user-interface and the FPGA is done with UDP/IP.

This connection can be implemented in different ways. There are two approaches how to send and receive the data between the FPGA and the user-interface:

In the first approach a similar design was chosen. The communication is also done with the Processing System of the ARM-Processor. The communication can either be TCP/IP (Figure 2) or UDP/IP (Figure 3). In both cases, the implementation for the FPGA is the same. Due to faster possible data rates (MAC interface supports 1GBit/s) and since data packet loss is acceptable, UDP/IP is for streaming data to the workstation more suitable. To use the PS, driver files are necessary to enable different functionalities for the communication. The ARM-Processor will execute these driver files, which sets up a UDP echo server to establish a connection between the workstation and the PS. After receiving the control information, the PS will write the control data into the RAM. Via the GPIO port, the PL can be enabled to read the data with the AXI DataMover [3] from the RAM. The Debug-Core extracts the different parameters out of the AXIS [2] data stream to start the debugging process with the chosen parameters. The sampled signal data is written with the AXI DataMover into the RAM. Each time, a certain number of samples was written into the RAM, an interrupt is raised. This number can be adjusted in the PS. The PS reads out the signal data and sends it to the workstation. The AID user-interface will handle the received data for monitoring and optional for logging the data into a csv file.

[10] Processing System see Chapter 4.2 Processing System

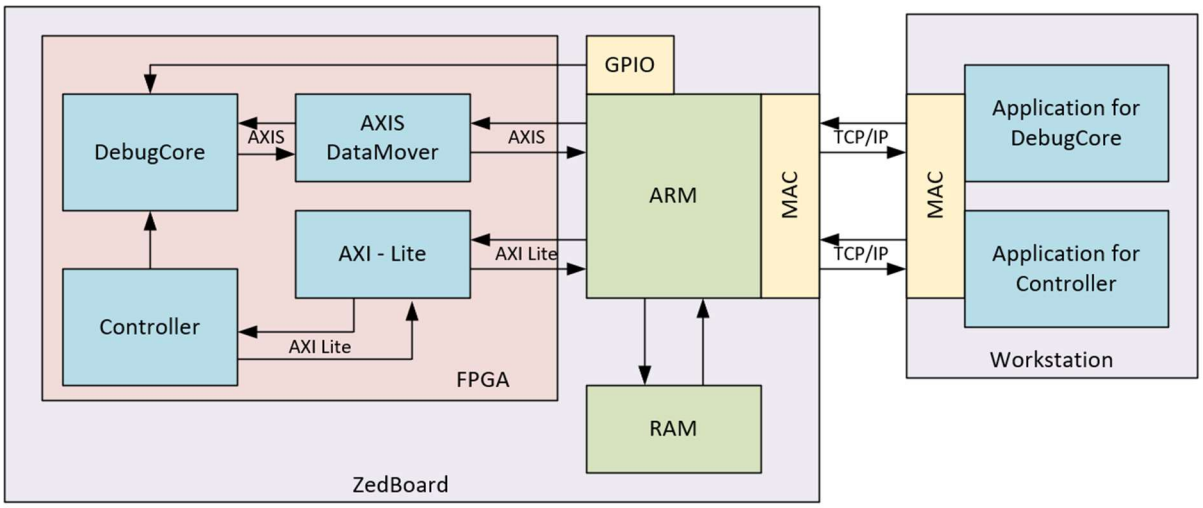


FIGURE 2: COMMUNICATION WITH THE PROCESSING SYSTEM AND TCP/IP

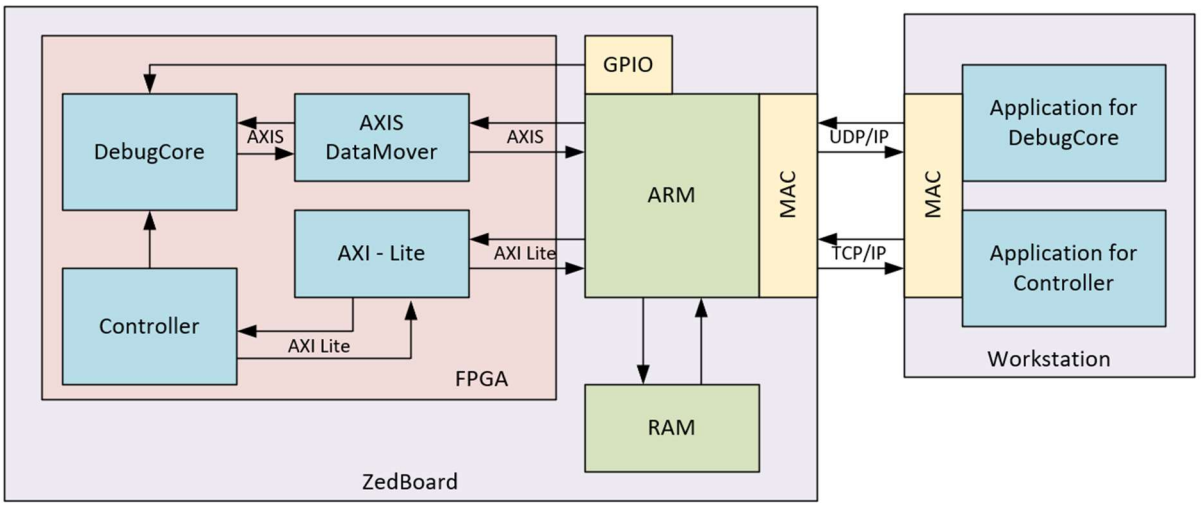


FIGURE 3: COMMUNICATION WITH THE PROCESSING SYSTEM AND UDP/IP

The second approach is, to avoid the Processing System for the communication. To do that, the AXI-Ethernet [4] IP core from Xilinx is used, as shown in Figure 4. This core converts the AXIS data stream into data for the Ethernet transeiver, which builds and sends the UDP/IP package to the workstation. To use the AXI-Ethernet core, a licence for the Xilinx TEMAC [5] (Tri-Mode Ethernet MAC) core is necessary. Due to the missing licence this approach was not possible.

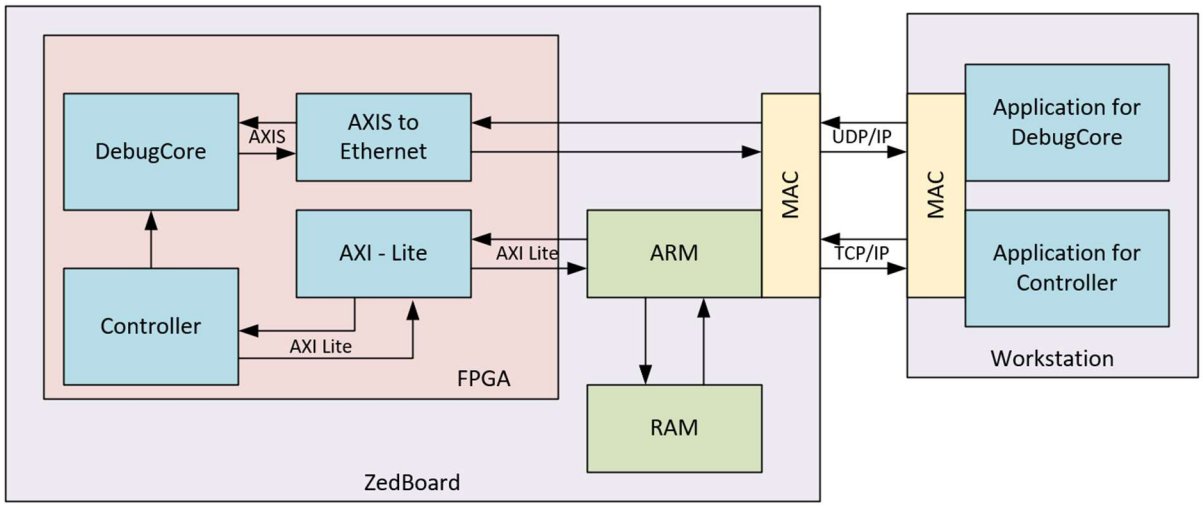


FIGURE 4: COMMUNICATION WITH THE AXI-ETHERNET IP CORE

Therefore, the decision was made to implement the communcation with the PS of the ARM-Processor and a UDP/IP echo server. Currently, the Controller IP core communicates with the Controller LabVIEW user-interface via the ARM-Processor and TCP/IP. Therefore, the driver files must be extended to include the UDP/IP communication between the Debug-Core and the Debug-Core user-interface. Furthermore, the PS has to enable the read operation from the RAM (PL side) to start the debugging process with the adjusted debugging parameters and a interrupt handler is necessary when the sampled signal data is written into the RAM and ready to be sent to the user-interface.

Maybe in the future, the AXI-Ethernet module will be used to avoid the communication via the ARM-Processor and the PS. The Processing System is slowly and when the Controller IP core is communicating a lot with the Controller user-interface, the Processor utilization increases. This can lead to performance problems of the Controller and the Debug-Core.

4 DEBUG-CORE ON THE FPGA

4.1 STRUCTURE OF THE DEBUG-CORE ON THE FPGA

The Debug Core on the FPGA is structured into different submodules, shown in Figure 5. These modules together provide the functionality on the FPGA.

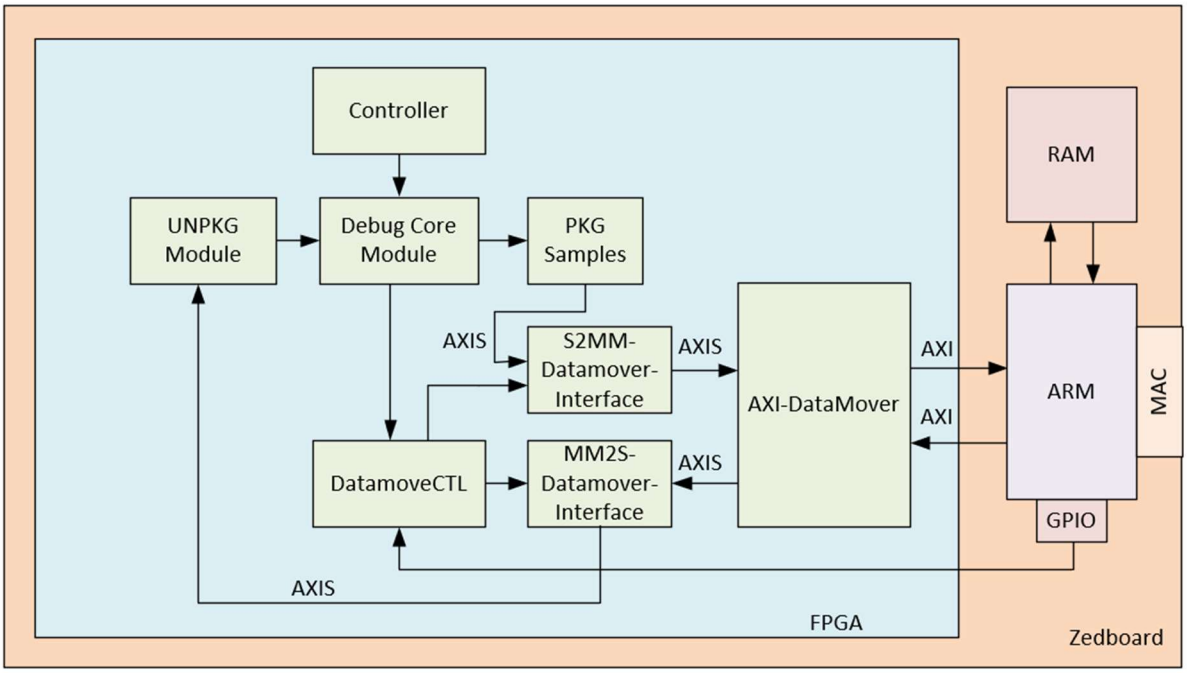


FIGURE 5: DEBUG-CORE STRUCTURE ON THE FPGA

To start the debugging process, the Processing System waits for the arrival of the control information from the user-interface. The PS writes the received data into the RAM and it accesses the AXI_GPIO ports to activate the data transfer from the RAM to the AXI DataMover. The AXI DataMover routes the control data through the MM2S-DataMover-Interface[11] to the UNPKG-Module[12] of the AID. The MM2S-Datamover-Interface block is an interface, which gives the AXI DataMover the control information for the read operation. Furthermore, it checks if the transfer from the RAM was successful. The UNPKG-Module checks, if the package type is for starting the Debug-Core or for resetting it. It also splits the AXIS data stream into the different parameters, which are needed to control the Debug-Core Module. The Debug-Core Module itself is the part, which handles the sampling rate, the numbers of samples, and waits for the right trigger event. It starts the debugging process and returns the sampled data. The PKG-Samples module[13] converts the sampled data to an AXIS data stream and sends it with the help of the DatamoveCTL block to the AXI DataMover. The AXI DataMover writes the sampled signal data into the RAM. The PS reads the data from the RAM and sends it to the workstation, where the user-interface can monitor the signal data.

[11] MM2S-DataMover-Interface stands for Memory Mapped to Stream DataMover Interface, see chapter 4.4
[12] UNPKG-Module stands for Unpackage Module, see chapter 4.7
[13] PKG-Samples Module stands for Package Samples Module, see chapter 4.8

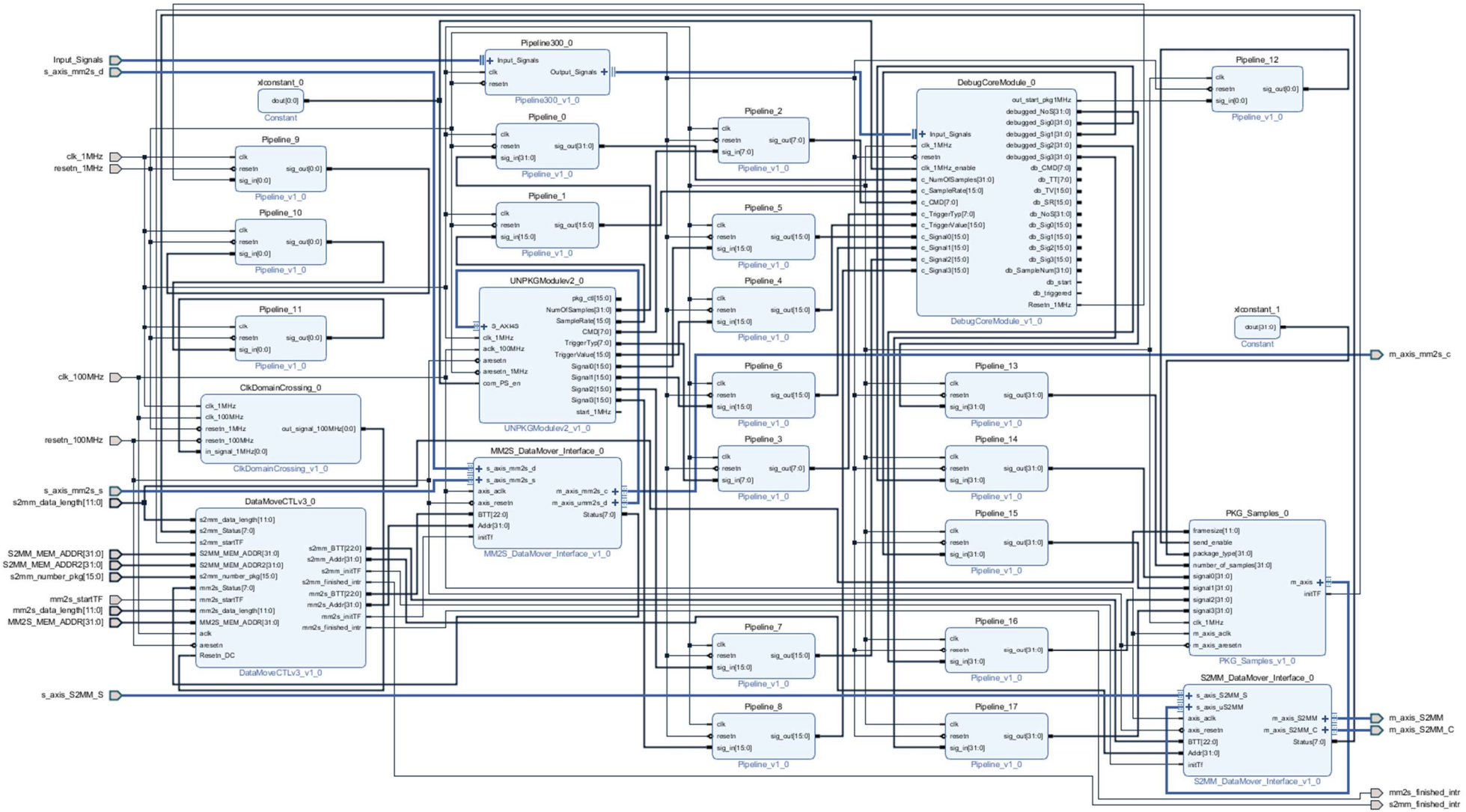


FIGURE 6: INTERNAL STRUCTURE OF THE AID IP CORE

Figure 6 shows an overview of the AID IP core with all the sub modules. The S2MM-Datamover-Interface[14] and the MM2S-Datamover-Interface provide the interfaces to control the AXI DataMover with commands from the DatamoveCTL module[15]. The DatamoveCTL module counts the samples, which are written into the RAM. If the number is reached, which was set by the PS at the initialization, an interrupt occurs and the PS handles the interrupt with the corresponding interrupt handler and reads out the sampled data from the RAM, builds and sends the UDP package to the workstation. Currently, 32 samples are collected in the RAM for the UDP transmission. One sample corresponds to one data package with the package type, sample number and the 4 signal values, each 32 bit long[16]. 32 of these data packages are put together to build the UDP payload. Therefore, the UDP package payload is 768 bytes.

4.2 IP CORE PROCESSING SYSTEM

The Processing System [1] (PS) offers the whole functionality of the ARM-Processor. Some of the functionalities are interfaces to peripherals, interfaces to memory, clocks, high performance ports and DMA, shown in Figure 7. Functionalities can be activated to use them in the design. The AID requires a proper PS-PL (Processing System to Programmable Logic), DDR, clock and interrupt configuration.

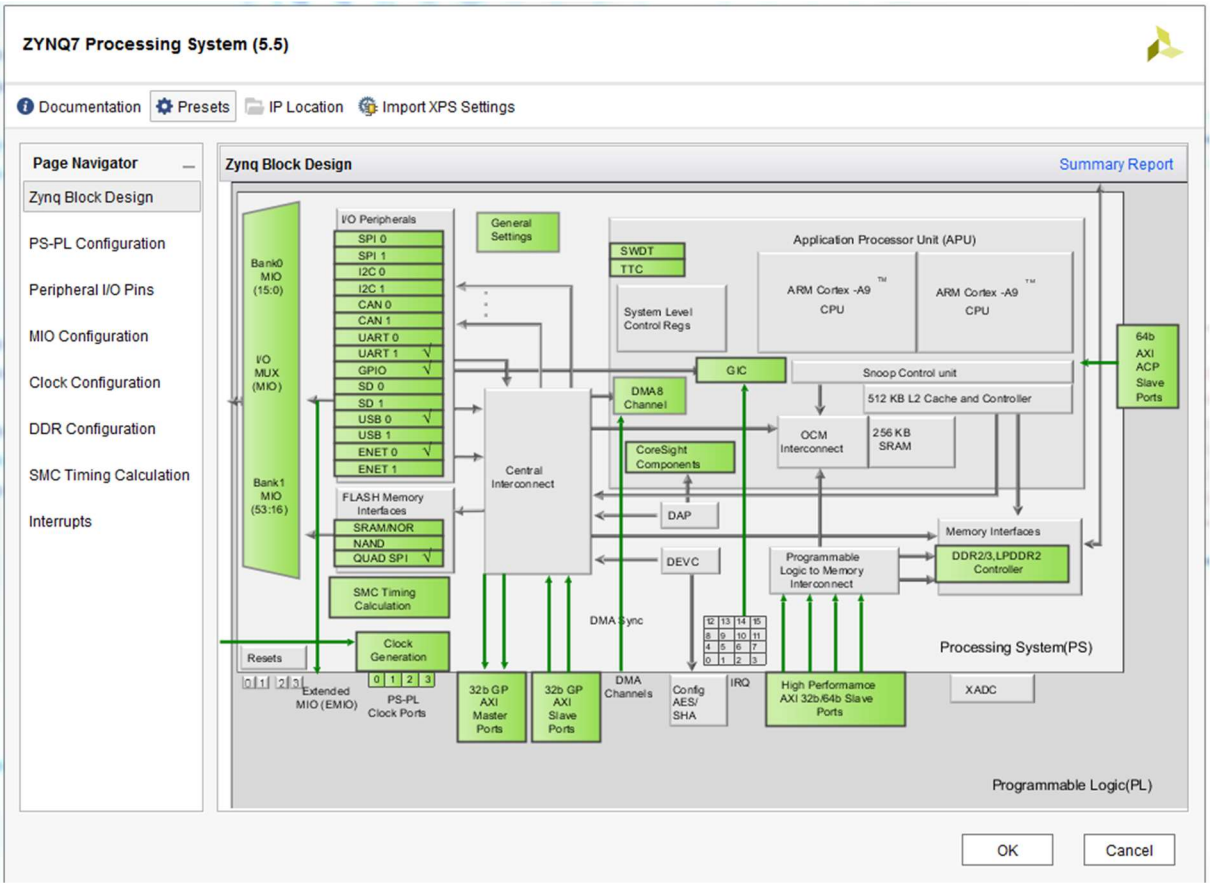


FIGURE 7: BLOCK DESIGN OF THE ZYNQ PROCESSING SYSTEM

[14] S2MM-DataMover-Interface stands for Stream to Memory Mapped DataMover Interface
[15] See chapter 4.6, IP core DataMoveCTL
[16] See chapter 11, UDP connection

The Processing System provides the clocks for the different modules of the Debug-Core. In this design, the PL Fabric clocks with 100 MHz and 1 MHz are used, shown in Figure 8. The Debug-Core, UNPKG and PKG Samples modules use the 1 MHz clock. This is also the maximum sampling frequency for the debugging process. The UNPKG and PKG Samples modules additionally use the 100 MHz clock for the AXI-Stream transfers. This frequency is necessary to split the configuration data into the different parameters for the debugging process and to build the AXIS data stream to save the sampled signal data into the RAM. The different AXI modules use the 100 MHz clock.

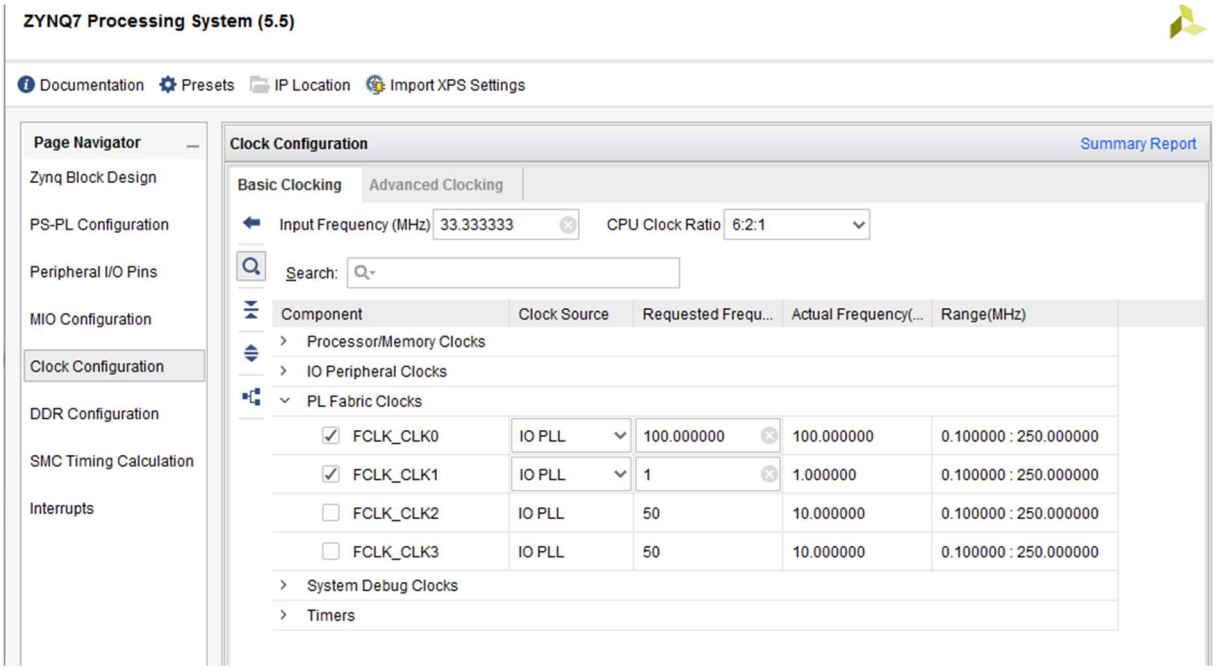


FIGURE 8: CLOCK SETTINGS OF THE ZYNQ PROCESSING SYSTEM

To access the Programmable Logic (PL) from the PS, AXI_GPIO [6] ports are used. Therefore, the AXI GP master interface must be enabled in the PS-PL section. Figure 9 shows the M_AXI_GP0 output port. With the AXI GP master interface, the AXI to GPIO blocks can be addressed and the signals can be set with the driver software. Parameters of the AID, like memory addresses, data length and number of samples for the UDP package are set with this interface. Also, the start command to read the control information from the RAM is done with an AXI_GPIO port.

To access the DDR RAM from the PL, the AXI High Performance Slave port (S_AXI_HP0) must be activated. This is done in the DDR section. With the S_AXI_HP0 port, data can be transferred between the AXI DataMover and the RAM. The control information is read from the RAM and the sampled signal data are written into the RAM.

To signal the PS that data is available, an interrupt is raised. To use the interrupts, Fabric Interrupts are activated. The interrupt is set by the DataMoveCTL block and is routed into the interrupt interface, IRQ_F2P, of the PS, shown in Figure 9.

The first interrupt (mm2s_finished_intr) is set, when the command data was successfully transferred from the RAM to the AXI DataMover. The interrupt handler resets the signal, which enabled the read operation. At that point, no read operations are necessary. The signal to start the read operation will be set again, when the PS received new control information from the user-interface.

The second interrupt (s2mm_finished_intr) is set, when the number of samples for the UDP package is reached and all the data is successfully written into the RAM. The interrupt handler of the s2mm_finished_intr is called. The PS reads the signal data from the RAM and sends it with a UDP package to the workstation.

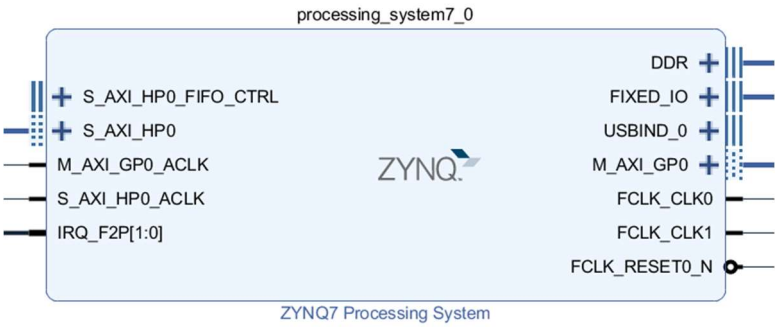


FIGURE 9: ZYNQ PROCESSING SYSTEM

Two Fabric clocks are used with 100 MHz and 1 MHz. The PS also offers a reset. This reset is active low. To use this reset, a Processor System Reset [7] block is used (Figure 10). It uses the clock and the FCLK_RESET0_N signal to generate the peripheral reset signal, which is also active low. For both clocks, Processor System Reset modules are used.

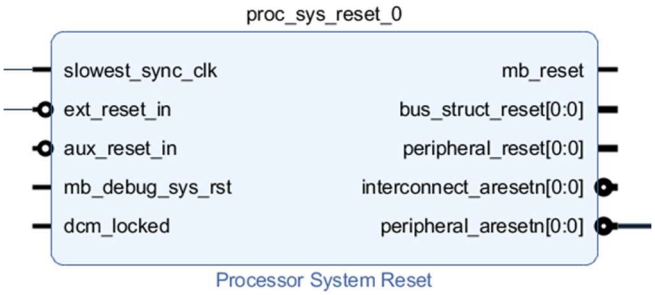


FIGURE 10: PROCESSOR SYSTEM RESET BLOCK

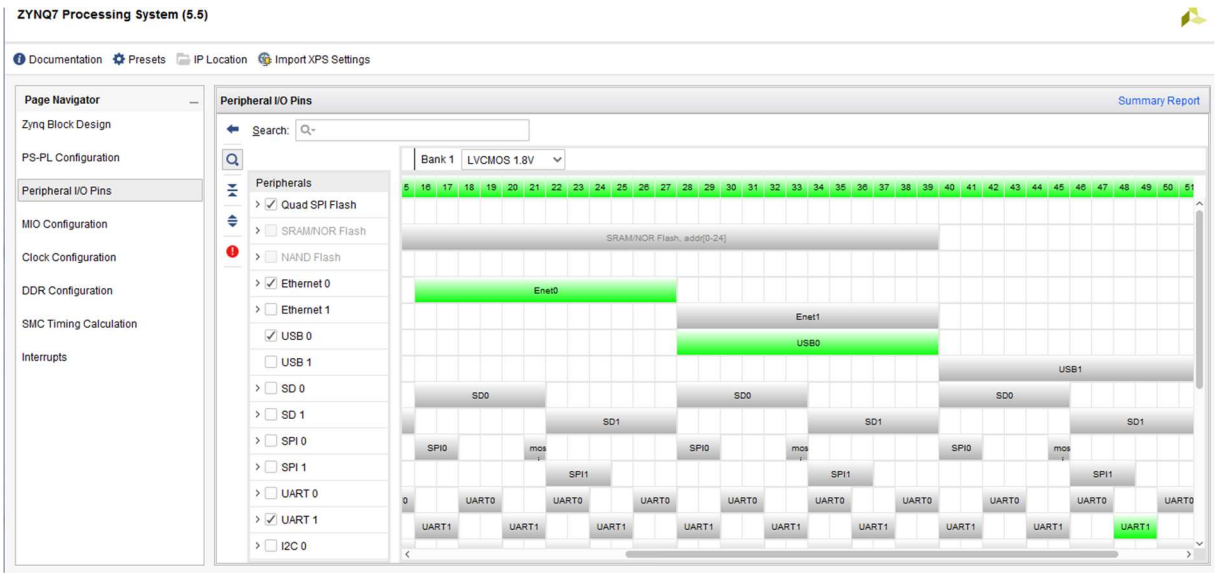


FIGURE 11: ZYNQ PROCESSING SYSTEM I/O PERIPHERALS

To use the integrated Ethernet interface of the ARM, the Ethernet I/O peripheral must be activated. Also the UART1 is activated to debug some signals from the design (ILA core with JTAG), to verify the correct work of the Debug-Core. USB0 is activated but not used. Figure 11 shows the activated communication interfaces.

4.3 IP CORE AXI-DATAMOVER

The AXI DataMover [3] is an IP-core provided by Xilinx. It is able to write and read AXIS [2] data streams to and from the RAM via the AXI4 interface. Commands are sent to the AXIS command slave interfaces of the IP-core to initialize the transfers. For writing data into the RAM, the Stream to Memory Mapped (S2MM) ports are used and to read data from the RAM, the Memory Mapped to Stream (MM2S) ports are used. The command data contains information like start memory address, bytes to transfer and other information. After the data transfer, status information is returned with the STS Master interfaces and with error signals. The AXI DataMover block requires clock and reset signals. All of the AXI4 interfaces use the 100 MHz PL Fabric Clock of the Processing System and the associated resetn signal. The error signals are not used for further processing but the status stream is used for checking if the data transfer was successful. The AXI DataMover block is shown in Figure 12.

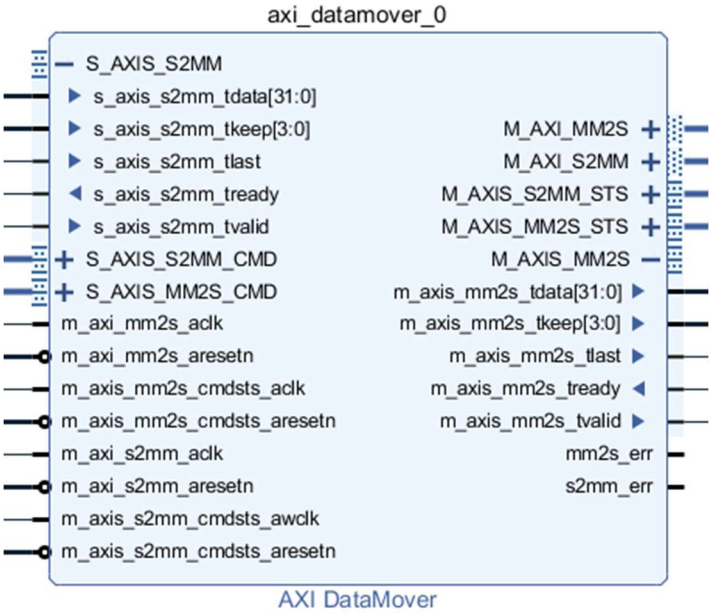


FIGURE 12: AXI DATAMOVER WITH INPUTS AND OUTPUTS

4.3.1 STREAM TO MEMORY MAPPED

To write data streams into the RAM, the Stream to Memory Mapped (S2MM) ports are used. There is an interface for S2MM command information (S_AXIS_S2MM_CMD), which contains the bytes to transfer (BTT), the type of AXI4 access, the start memory address, DRE Stream Alignment and Realignment, End of Frame and Command TAGs.

To start the transfer, the PKG_Samples block sets the initTF signal. The DataMoveCTL block immediately sends the transfer settings to the S2MM_DataMover_Interface, which builds the command data stream to initialize the data transfer into the RAM. Then, the AXIS data stream is sent to the S_AXIS_S2MM interface of the AXI DataMover and furthermore with the AXI bus to the AXI High Performance Slave interface of the Processing System (S_AXI_HP0). The data is forwarded into the DDR RAM. After the data transfer, a status is returned via the M_AXIS_S2MM_STS interface. If the transfer was successful, the status value is 0x80 and the DataMoveCTL block sets the interrupt or increases the internal sample counter for further transfers.

4.3.2 MEMORY MAPPED TO STREAM

To read data streams from the RAM, the Memory Mapped to Stream (MM2S) ports are used. There is an interface for MM2S command information (S_AXIS_MM2S_CMD), which contains the bytes to transfer (BTT), the type of AXI4 access, the start memory address, DRE Stream Alignment and Realignment, End of Frame and Command TAGs.

To start the transfer, the Processing System sets the mm2s_startTF signal of the DataMoveCTL block. This initializes the read operation with the proper settings for the AXI DataMover. Then, the data stream is read from the RAM with the AXI High Performance Port. The M_AXI_MM2S interface of the AXI DataMover receives the data, forwards it via the M_AXIS_MM2S interface to the user logic. After the data transfer, a status is returned via the M_AXIS_MM2S_STS interface. If the transfer was successful, the status value is 0x80 and the DataMoveCTL block sets the mm2s_finished_intr interrupt. The interrupt handler resets the mm2s_startTF signal. It will be set again with the PS, when new command information should be read from the RAM.

4.3.3 COMMAND INFORMATION FOR MM2S AND S2MM TRANSFERS

For the AID, the type of AXI4 access is set to 1 (INCR) and there is no use of Stream Alignment and Realignment. INCR automatically increases the memory address with the bytes to transfer. This setting is used for S2MM and MM2S transfers. The start memory address can be set with GPIO ports from the Processing System or with a generic in the DataMoveCTL block. The DataMoveCTL block also sets the BTT (Bytes to transfer) field for the command and sends this information to the DataMover_Interfaces to build the command data stream.

4.3.4 AXI DATAMOVER IP SETTINGS

The AXI DataMover can access the RAM with write and read operations. It can be configured for the appropriate use with its IP settings, as shown in Figure 13. The most important settings are, to enable MM2S and S2MM transfer, Memory Map Data Width, Stream Data Width, Maximum Burst Size and the Width of the BTT (Bytes to transfer) field.

Both transfer directions are enabled, both data width are set to 32 bits, maximum burst size is set to 16 and the BTT width is set to 23. The chosen settings for the AXI DataMover handle the data transfer between the AID and the RAM.

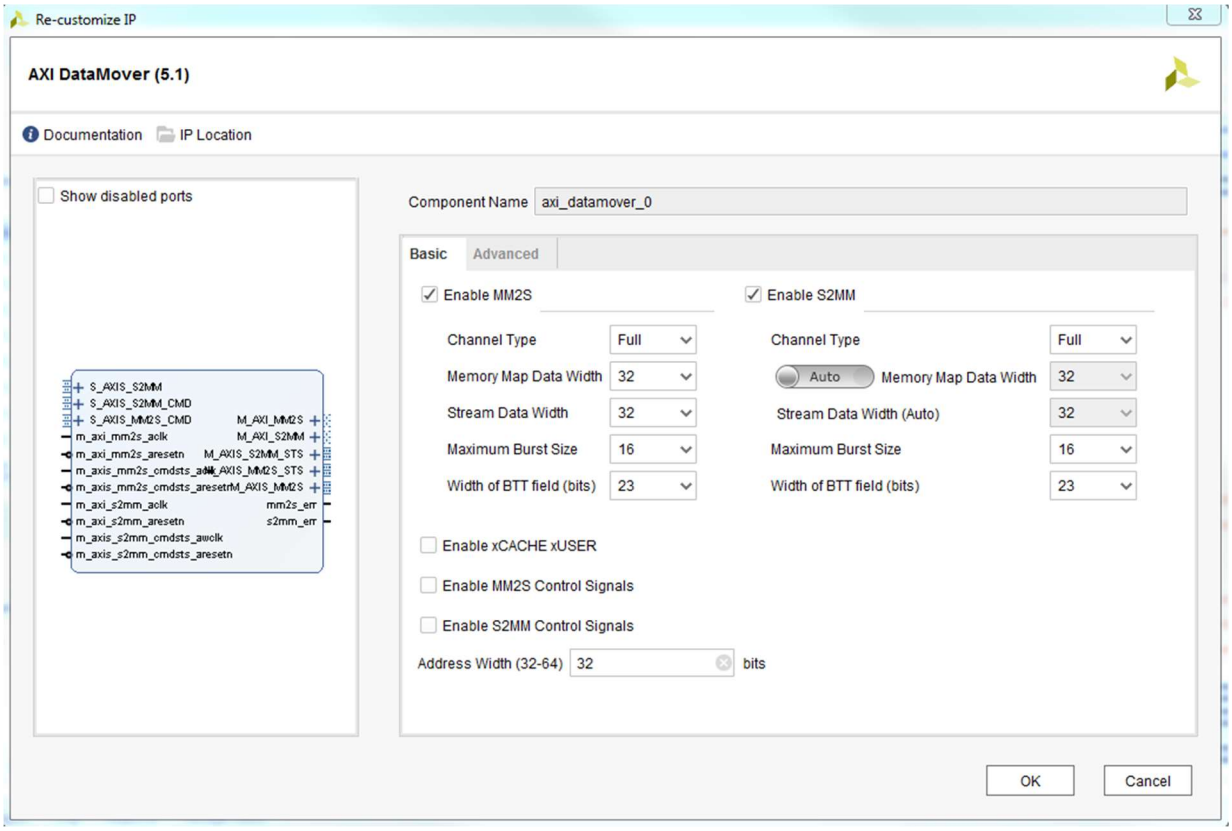


FIGURE 13: AXI DATAMOVER IP SETTINGS

4.4 IP CORE MM2S_DATAMOVER_INTERFACE

This IP core was provided by my college Stephan Hochmüller from AVL List GmbH. It connects the AXI DataMover with the DataMoveCTL block. The DataMoveCTL block sets the BTT (Bytes to Transfer), the Addr (memory address) and the initTf (initialize transfer) to send the command data with the m_axis_mm2s_c[17] interface to the AXI DataMover, which starts the read operation with the adjusted settings, from the RAM. The control data for the debugging process are routed from the AXI DataMover via the AXIS-interfaces s_axis_mm2s_d[18] and m_axis_umm2s_d[19] to the user logic (UNPKGModule). With the AXIS interface s_axis_mm2s_s[20], the transfer status stream is decoded to a Status signal, which is processed by the DataMoveCTL block. The status signal gives information about the read operation from the RAM. The IP core is shown in Figure 14. An overview of the inputs and outputs of this IP core is shown in Table 6 and Table 7 in the Appendix. The data width of the AXIS interface and the BTT can be changed in the property window of the IP core.

[17] M_axis_mm2s_c stands for Master AXIS Memory Mapped to Stream Command interface
[18] S_axis_mm2s_d stands for Slave AXIS Memory Mapped to Stream Data interface
[19] M_axis_umm2s_d stands for Master AXIS User Memory Mapped to Stream Data interface
[20] S_axis_mm2s_s stands for Slave AXIS Memory Mapped to Stream Status interface

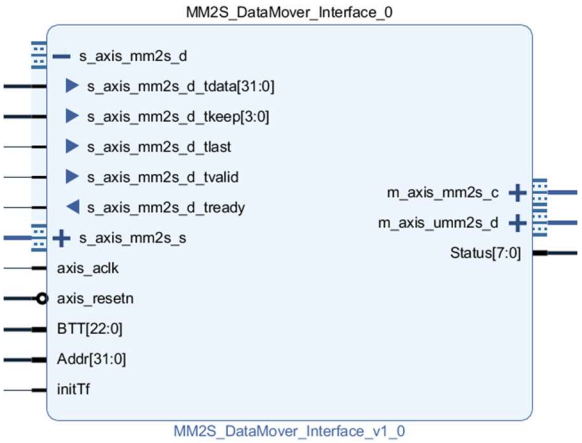


FIGURE 14: MM2S_DATAMOVER_INTERFACE IP CORE WITH INPUTS AND OUTPUTS

4.5 IP CORE S2MM_DATAMOVER_INTERFACE

This IP core was provided by my college Stephan Hochmüller from AVL List GmbH. It connects the AXI DataMover with the DataMoveCTL block. The DataMoveCTL block sets the BTT, Addr and initTf to send the command data with the m_axis_S2MM_C[21] interface to the AXI DataMover, which starts the write operation with the adjusted settings, into the RAM. The signal data is routed from the PKG Samples block via the interfaces s_axis_uS2MM[22] and m_axis_S2MM[23] to the AXI DataMover. The AXI DataMover returns the transfer status stream to the s_axis_S2MM_S[24] interface, which is decoded into a Status signal. The information of the Status signal is processed by the DataMoveCTL block. The IP core is shown in Figure 15. The data width of the AXIS interface and the BTT can be changed in the property window of the IP core. An overview of the inputs and outputs of this IP core is shown in Table 8 and Table 9 in the Appendix.

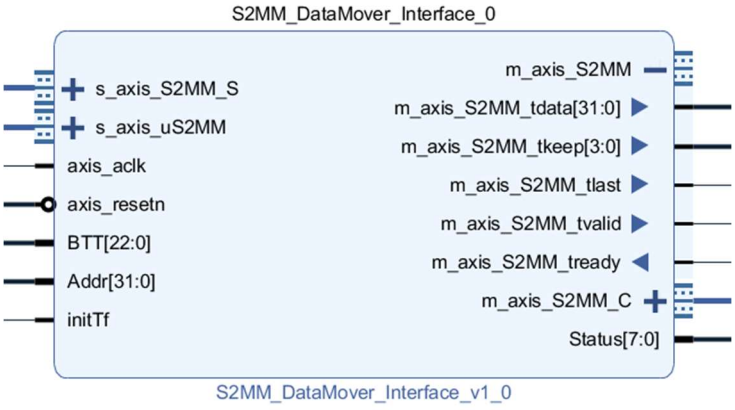


FIGURE 15: S2MM_DATAMOVER_INTERFACE INPUTS AND OUTPUTS

[21] M_axis_S2MM_C stands for Master AXIS Stream to Memory Mapped Command Interface
[22] S_axis_uS2MM stands for Slave AXIS User Stream to Memory Mapped Interface
[23] M_axis_S2MM stands for Master AXIS Stream to Memory Mapped Interface
[24] S_axis_S2MM_S stands for Slave AXIS Stream to Memory Mapped Interface

4.6 IP CORE DATAMOVECTL

The DataMoveCTL IP core coordinates the data transfer between the AXI DataMover and the user logic. It was developed by me during the implementation of the Debug-Core. There are input ports to set up the data transfer in S2MM and MM2S direction. The DataMoveCTL block is shown in Figure 16. An overview of the inputs and outputs of this IP core is shown in Table 10 and Table 11 in the Appendix.

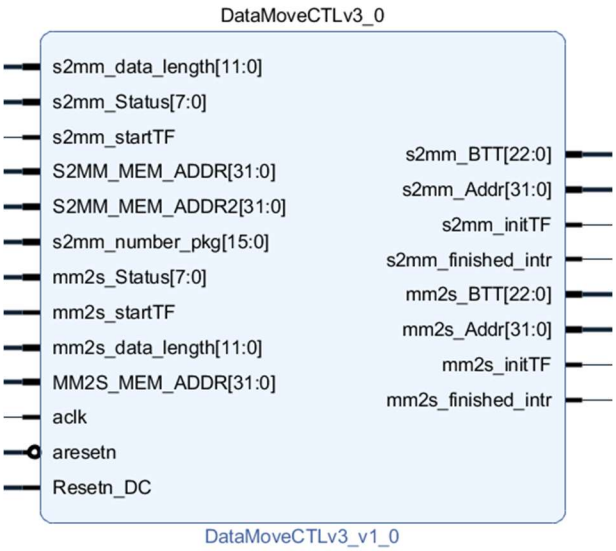


FIGURE 16: DATAMOVECTL WITH INPUTS AND OUTPUTS

The DataMoveCTL block contains different IP settings. The width of the memory addresses and of the Bytes to transfer is always set in the IP Settings. The address width is set to 32 bits and BTT is set to 23 bits. The other options can either be set with the IP Settings or with AXI GPIO [6] ports from the Processing System. Currently, the control with the Processing System (C_PS_CONTROL) is active and the memory addresses and the C_NUMBER_PKG is defined with the PS and the GPIO ports. The C_NUMBER_PKG defines, how many data packages are collected for the UDP payload, currently it is set to 32 packages. An overview of the IP settings is shown in Table 12 in the Appendix.

4.6.1 INITIALIZE MM2S TRANSFER

To initialize the transfer from the RAM to the user logic to start the whole debugging process, the MM2S ports have to be addressed. Therefore, the start memory address (MM2S_MEM_ADDR) and the data length in bytes (mm2s_data_length) have to be set either with the Processing System or in the IP settings. The MM2S transfer is initialized by the PS. The PS sets the AXI_GPIO port, which is connected to the mm2s_startTF input. When the mm2s_startTF signal is set, the DataMoveCTL block sends the proper command information via the signals mm2s_BTT, mm2s_Addr and mm2s_initTF to the MM2S_DataMover_Interface. The MM2S_DataMover_Interface builds the command data and sends it to the AXI DataMover. The AXI DataMover starts the reading operation. When the data was transferred, the MM2S_DataMover_Interface gets the status stream back, decodes it into the Status signal and sends the status to the DataMoveCTL mm2s_Status input. If the transfer was successful, the status value is 0x80 and the interrupt (mm2s_finished_intr) for the MM2S transfer is set. The Processing System catches and handles the interrupt. The PS resets the mm2s_startTF signal to stop further transfers. When the PS receives data from the user-interface, the UDP data is stored in the RAM and the mm2s_startTF signal is set again, to start the next read operation.

4.6.2 INITIALIZE S2MM TRANSFER

To initialize the data transfer from the user logic (PKG_Samples) to the RAM, the S2MM ports have to be addressed. The PKG_Samples block generates the s2mm_startTF signal. The s2mm_data_length is the length of the data in Bytes, which will be written into the RAM. Both S2MM memory addresses and the number of samples can be set either with the port signals or in the IP Settings. To enable the use of the port signals, C_PS_CONTROL has to be enabled. This allows setting up the DataMoveCTL block with the Processing System and the AXI GPIO ports.

One of the two S2MM start memory addresses is used to write the signal data into the RAM. The C_NUMBER_PKG/s2mm_number_pkg defines the number of samples, which are stored into the RAM until the s2mm_finished_intr is set. After each data transfer from the AXI DataMover to the RAM, the S2MM_DataMover_Interface receives the status stream and returns the Status to the DataMoveCTL block. If the transfer was successful, the Status value is 0x80 and the internal counter increases. When the C_NUMBER_PKG/ s2mm_number_pkg is reached, the interrupt signal (s2mm_finished_intr) is set. The Processing System handles the interrupt, reads the stored data from the RAM, and builds the UDP package. The package is transmitted via the integrated Ethernet interface of the ARM-Processor to the workstation. In addition, the S2MM memory address are changed from S2MM_MEM_ADDR to S2MM_MEM_ADDR2, when the internal sample counter reached the value of C_NUMBER_PKG/s2mm_number_pkg.

This mechanism avoids concurrent read and write operations to and from the RAM. During the reading process of the Processing System, the signal data is written into the RAM with the second memory address. If the C_NUMBER_PKG/s2mm_number_pkg is reached again, the memory address is changed back to S2MM_MEM_ADDR. The memory address switch is shown in Figure 17

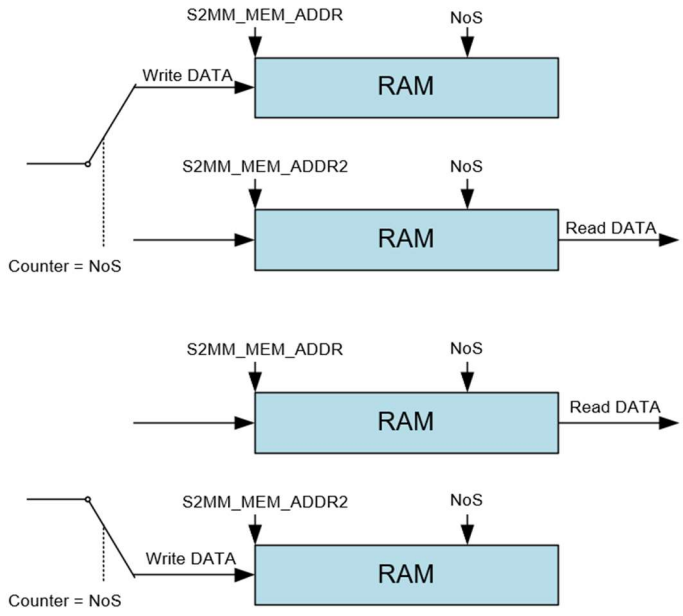


FIGURE 17: DATAMOVECTL MEMORY ADDRESS SWITCHING

Figure 18 shows the simulation of the DataMoveCTL block during the MM2S transfer. The MM2S start memory address was set via the GPIO port and the Processing System. The C_PS_CONTROL is enabled (red), therefore the corresponding values from the input ports are used (orange) to set up the transfer from the RAM. With the mm2s_initTF signal (green at 1, 330 ns) the transfer from the RAM is initialized. When the transfer is successful, the value of the mm2s_Status signal is 0x80 (violet) and the finish interrupt is set (blue). This is shown at 1,399.000 ns.

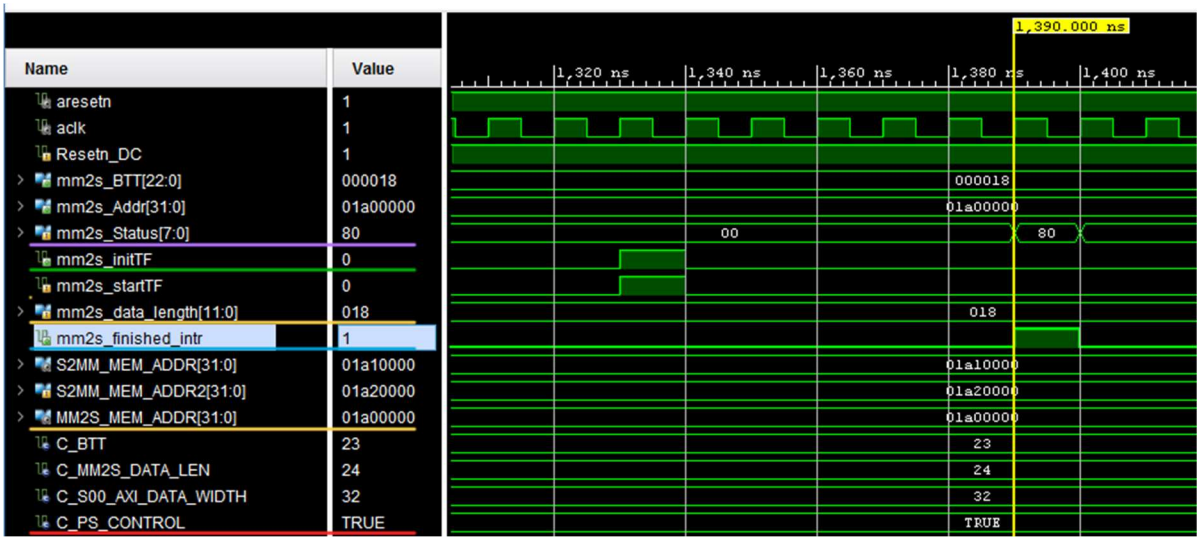


FIGURE 18: SIMULATION OF DATAMOVECTL DURING THE MM2S DATA TRANSFER

Figure 19 shows the simulation of the DataMoveCTL block after a successful S2MM data transfer. At 11,190.000 ns, the data transfer was successful, the transfer status is 0x80 (violet). The signal data were correctly written to the memory address 0x01A102E8 of the RAM. The interrupt is set (blue) because the number of collected samples for the UDP payload is reached. The internal sample counter is reset (red) and the memory address switch is done (orange). The new memory address is set to 0x01A20000. During the next write operations, the PS is able to read the previous signal data from the RAM without a conflict.

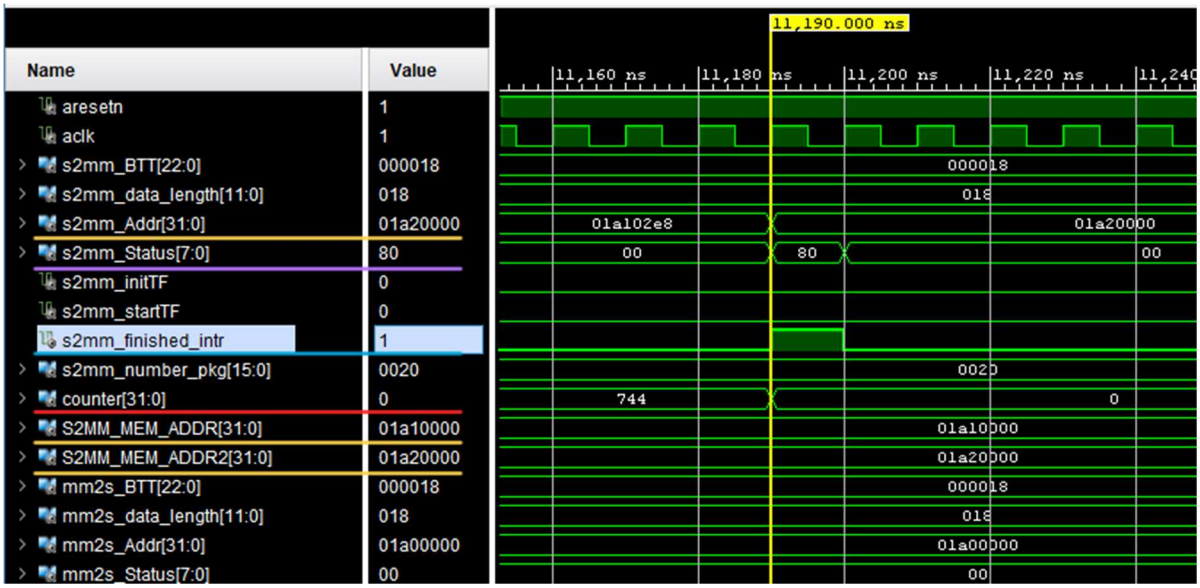


FIGURE 19: SIMULATION OF DATAMOVECTL AFTER A SUCCESSFUL S2MM DATA TRANSFER

Figure 20 shows also shows the simulation of DataMoveCTL with a successful S2MM data transfer. At 10,500 ns, the status signals a successful transfer with the value 0x80 (violet). No interrupt is set. The number of the collected samples for the UDP payload is not reached and the internal counter (red) is increased. Also the memory address s_2mm_Addr (orange) is updated. At 11,190.000 ns, the memory address switch (orange) is shown. The collected number of samples for the UDP payload is reached and the memory address is switched to S2MM_MEM_ADDR2 (orange). After the memory switch, the sample collection starts again.

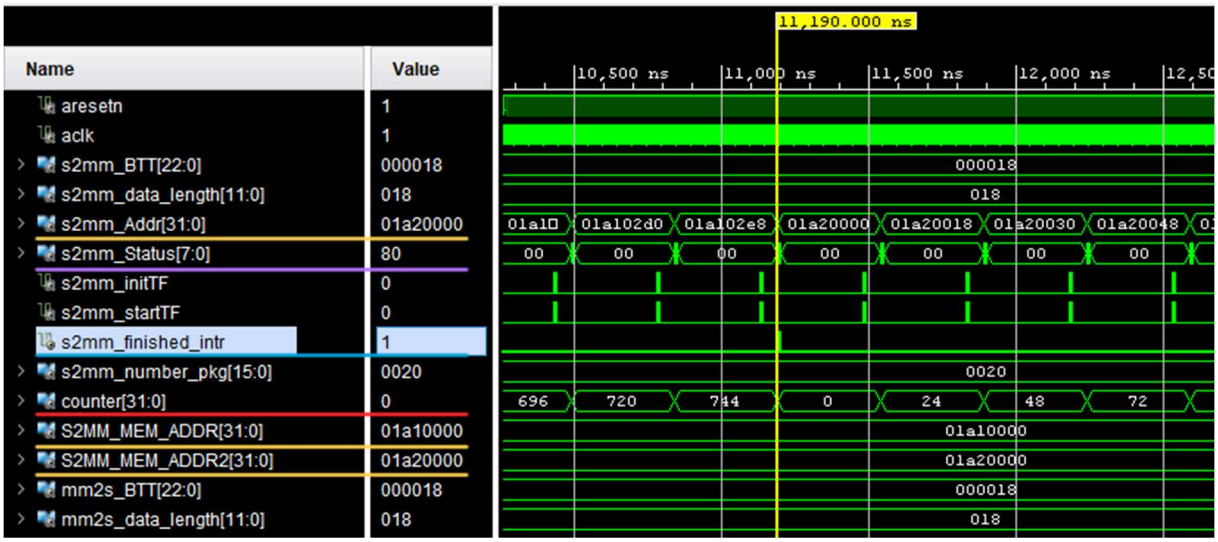


FIGURE 20: SIMULATION OF DATAMOVECTL WITH S2MM TRANSFER AND INTERRUPT

4.7 IP CORE UNPKGMODULE

The UNPKGModule decodes the AXIS data stream, which is sent from the AXI DataMover via the MM2S_DataMover_Interface to the UNPKGModule. This module was developed by me during the implementation of the Debug-Core. The tready signal of the AXIS slave interface (S_AXI4S) is always high (ready to receive data). The com_PS_en (communication Processing System enable) enables the communication with the PS. The outgoing signals are necessary for the debugging process with the different settings like commands[25] (CMD), numbers of samples, sample rate, trigger type, trigger value and the chosen signals to debug. The signals pkg_ctl (Package control) and start_1MHz were used for debugging. The module is shown in Figure 21. An overview of the inputs and outputs of this IP core is shown in Table 13 and Table 14 in the Appendix.

[25] Commands, see chapter 11 UDP connection

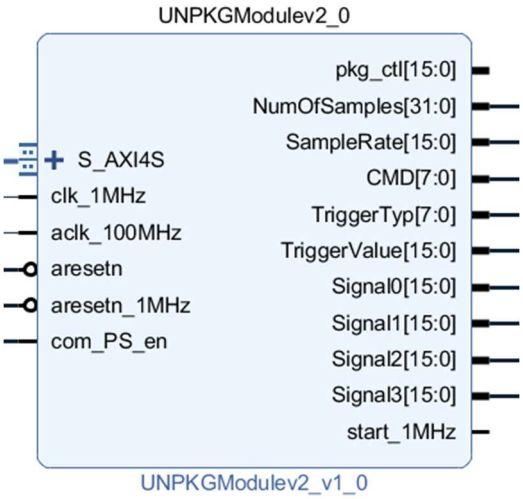


FIGURE 21: UNPKGMODULE WITH INPUTS AND OUTPUTS

4.7.1 INTERNAL STRUCTURE OF THE UNPKGMODULE IP CORE

Figure 22 shows the overview of the UNPKGModule. It is structured into the blocks UNPKG_UDP_CTL_Unit, UNPKG_UDP_DATA and UNPKG_UDP_Start. All three block together provide the functionality to map the AXIS data stream to the different control signals for the debugging process. The UNPKG_UCP_CTL block contains a state machine, which activates the different enable signals. The UNPKG_UCP_Data block uses the enable signals to route the incoming data to the corresponding output signals. The UNPKG_UDP_Start block, creates with the S_AXI4S_tlast signal, the 1 MHz start signal. The tlast signal of the AXIS data stream signals the end of the data transfer. It is set for only one clock cycle (100 MHz).

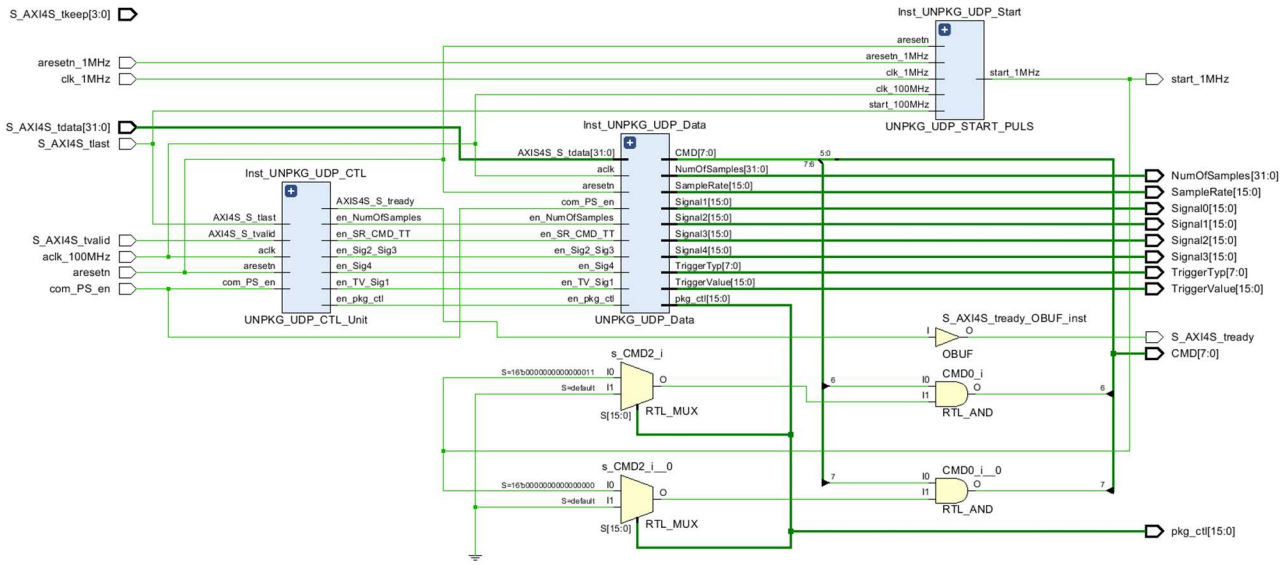


FIGURE 22: INTERNAL STRUCTURE OF THE UNPKGMODULE IP CORE

Figure 23 shows the simulation of the decoding process of the AXIS data stream. The AXIS data (orange) is mapped to the corresponding output signals (blue and violet). The AXIS data stream is sent with a frequency of 100 MHz (5,100 ns). Therefore, the data is converted into the 1 MHz clock domain for further use. At 6,000 ns, all values are assigned to the output signals. The start signal is generated (red). Also the CMD signal is assigned with the correct values depending on the control information from the user-interface. Here, the package type[26] was start debugging. Therefore, the start debugging bit is set of the CMD signal. After one clock cycle, this start bit is reset to 0 (at 7 ns), to avoid a restart of the debugging process. The com_PS_en signal enables the module for the communication with the ARM-Processor and the integrated Ethernet interface.

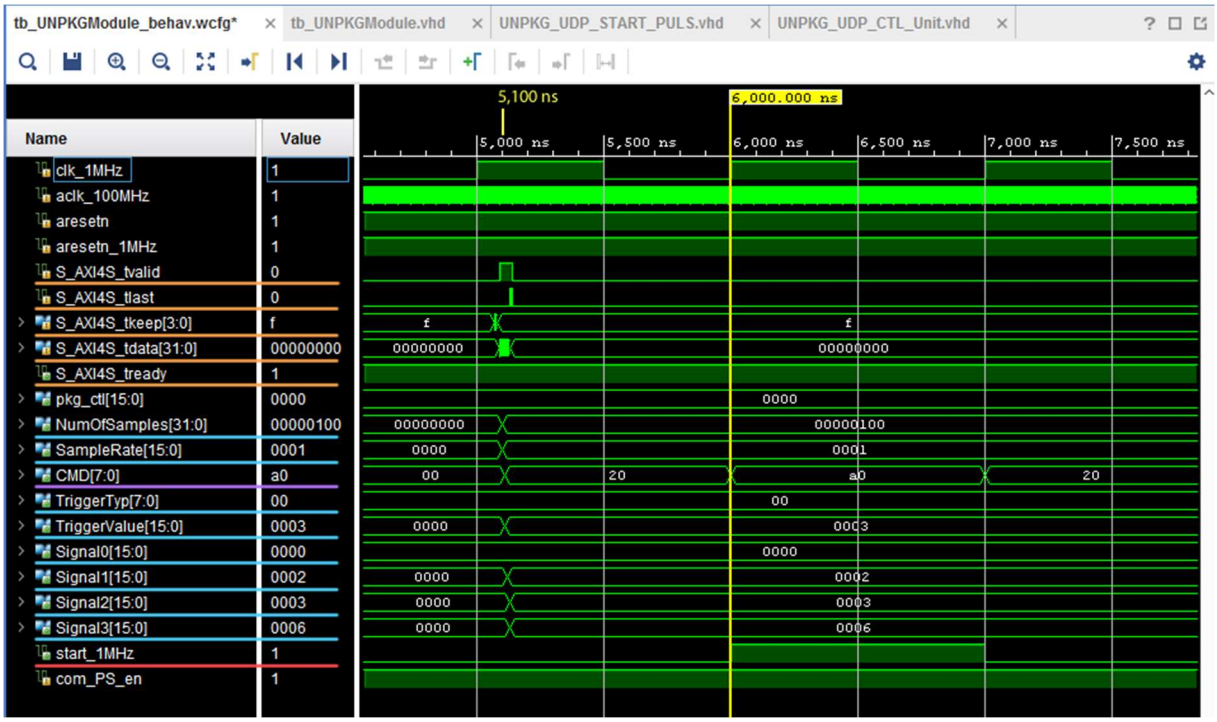


FIGURE 23: SIMULATION OF THE UNPKGMODULE PROCESSING THE AXIS DATA STREAM

4.7.2 UNPKG_UDP_CTL_UNIT BLOCK

This block counts the incoming data words of the AXIS data stream. The AXIS interface transfers 4 Bytes each clock cycle until the transfer is finished. Due to the AXIS transfer, the operating frequency of this block is 100 MHz. Depending on the counter, the enable signals are set to start the data mapping. The module is shown in Figure 24. A state machine is working inside this block. The state machine switches into the read data state (RD_DATA), when the tvalid signal of the AXIS interface is set (the tvalid signal starts the data transfer) In the read state, the internal counter is increased every clock cycle of the transfer, which enables the corresponding enable signals. With the tlast signal of the AXIS interface, the transfer is over and the state machine switches back into the IDLE state. The state machine is shown in Figure 25. The outputs of this block are the different enable signals to map the received AXIS data stream to the corresponding debugging parameters. It is also possible, that more than one debugging parameter is mapped by an enable signal, like the en_SR_CMD_TT signal. This signal enables the mapping of the sample rate, the CMD Byte and the trigger type. The tready signal for the AXIS interface is constantly set, which means the UNPKGModule is always ready to receive data. This block is designed, that only the configuration data (start debugging, stop debugging) from the user-interface is processed. The configuration data are always the same length. If wrong data is transmitted,

[26] Package Type, see chapter 11.1 Package Type Numbers

the Debug-Core is not able to start or reset the debugging process. An overview of the inputs and outputs of this IP core is shown in Table 15 and Table 16 in the Appendix.

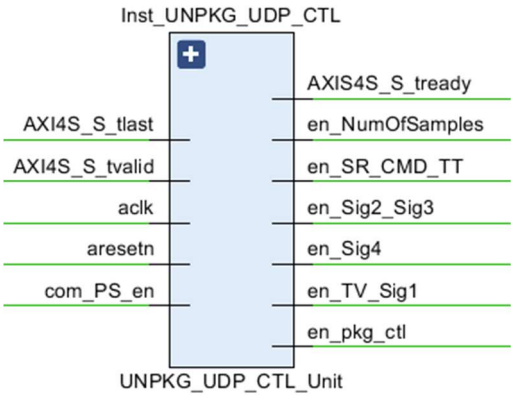


FIGURE 24: UNPKG_UDP_CTL_UNIT BLOCK

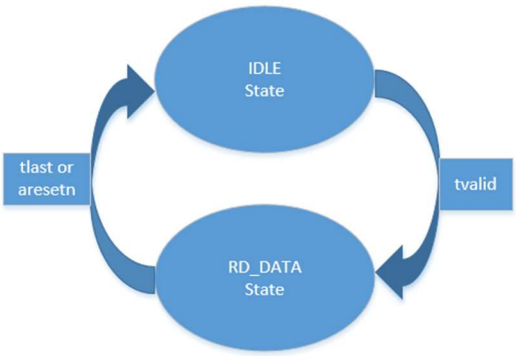


FIGURE 25: UNPKG_CTL_UNIT STATE MACHINE

Figure 26 shows the simulation of the state machine. Due to the set tready signal (orange), the AXIS interface is always ready to receive data. When the transfer starts, the tvalid signal gets set (orange at 30,430 ns). At this point, the state machine switches from the IDLA state into the RD_DATA state. At this point, the first 4 Bytes are transmitted with the AXIS interface. Therefore, the first enable signal is set, which is en_NumOfSamples (Number of Samples). The AXIS transfer is still active, as long as tvalid and tready is set. At each further clock cycle and ongoing data transfer, the internal counter increases and sets the other enable signal, en_SR_CMD_TT (sample rate, CMD, trigger type at 30,44 ns), en_TV_Sig1 (trigger value, signal 1 at 30,45 ns), en_Sig2_Sig3 (signal 2, signal 3 at 30,46 ns) and en_Sig4 (signal 4 at 30, 47 ns). The previous signals are reset again. The enable signal are marked blue. When the tlast signal is set, which means the data transfer is finished, the state machine switches back into the IDLE state (red at 30,47 ns).

FIGURE 26: SIMULATION OF THE STATE MACHINE OF THE UNPKG_CTL_UNIT BLOCK

4.7.3 UNPKG_UDP_DATA BLOCK

The UNPKG_UDP_Data block maps with help of the enable signals the data from the AXIS data stream to the right output signals. The AXIS data stream contains the information to start the debugging process with the adjusted parameters or reset the Debug-Core (stop debugging). This block also uses the 100 MHz clock due to the AXIS interface. The module is shown in Figure 27. The input signal en_pkg_ctl and the output signal pkg_ctl are not used. An overview of the inputs and outputs of this IP core is shown in Table 17 and Table 18 in the Appendix.

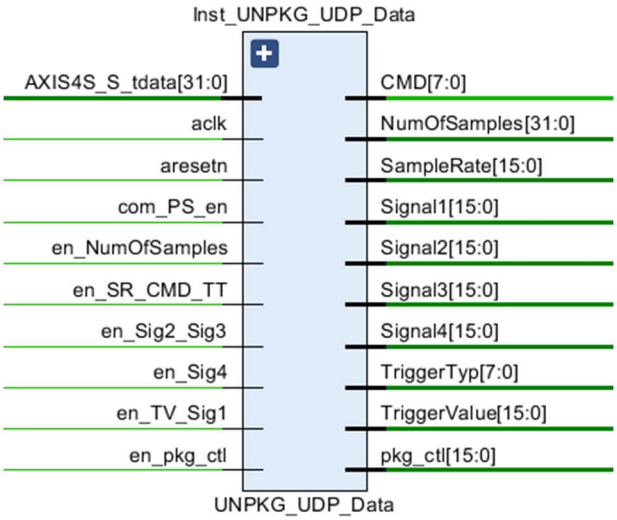


FIGURE 27: UNPKG_UDP_DATA BLOCK

Figure 28 shows the processing of the AXIS data stream. Due to the different enable signals, the values of AXIS data stream can be assigned to the different output signals. In one clock cycle the AXIS interface transfers 4 Bytes. At 5,090 ns, the en_NumOfSamples (green) is set, but no values are assigned to any

output signal. That is because, the first 2 Bytes is the package type[27] value and the value for the number of samples is a 32 bit value. Therefore, the NumOfSamples signal is assigned with a value, when the next 4 Bytes are received. This happens at 5,1 ns. The other 2 Bytes are assigned to the SampleRate signal (orange). At 5,11 ns, the en_TV_Sig1 signal enables the assignment for the CMD Byte, the trigger type and the trigger value (blue). At 5,12 ns, the values for the signals Signal1 and Signal2 are assigned and at the end the last two signals (violet) are set with their values. The assignments of all signals are delayed by one clock cycle.

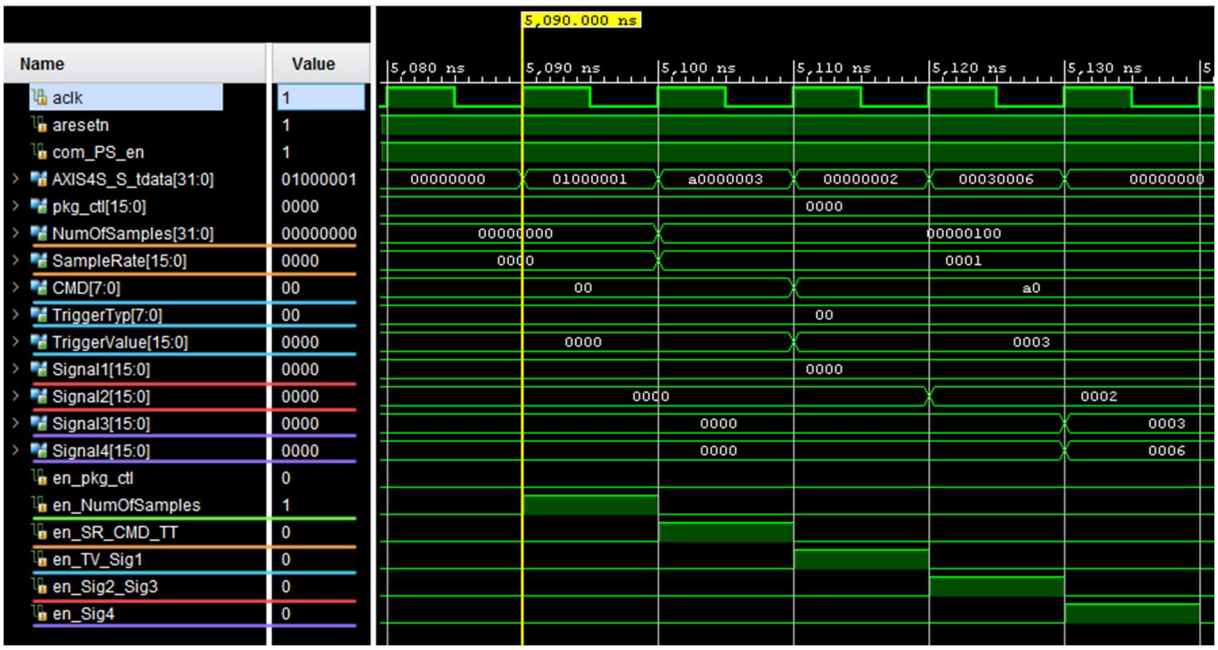


FIGURE 28: UNPKG_UCP_DATA SIMULATION OF PROCESSING THE AXIS DATA STREAM

4.7.4 UNPKG_UDP_START_PULS BLOCK

The UNPKG_UDP_Start_PULS block generates the 1 MHz start pulse to initialize the debugging process. The 1 MHz and 100 MHz clocks are used for this block. For this, the tlast signal of the AXIS interface is used. The tlast signal has a clock frequency of 100 MHz and with the detection of the falling and rising edge of the 1 MHz clock, the 1 MHz start pulse is created. Also the CMD_Reset signal is created this way, depending on the incoming package type[28]. Package type 0 (start debugging package) generates the start signal and package type 3 (stop debugging/reset) generates the reset/stop signal. The block is shown in Figure 29. An overview of the inputs and outputs of this IP core is shown in Table 19 and Table 20 in the Appendix.

[27] Package type of the UDP payload, see chapter 11.1 Package Type Numbers
[28] Package type, see chapter 11.1 Package Type Numbers

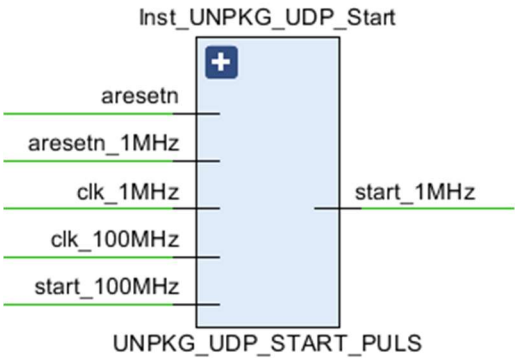


FIGURE 29: UNPKG_UDP_START_PULS BLOCK

Figure 30 shows the simulation of the UNPKG_UDP_Start_Puls block at the generation of the 1 MHz start signal. The input is the start_100MHz signal (blue), which is the tlast signal of the AXIS interface. This signal is only set for a 100 MHz clock cycle, when the AXIS data transfer is finished.

At 15,265 ns, the rising edge of the start_100MHz signal is detected and the s_puls signal (orange) is set. Due to the enabled s_puls signal, the signals start_1MHz (red) and s_en_fedge[29] (green) are set at 16,000 ns.

Due to the high s_en_fedge signal at the falling edge of the 1 MHz clock, the signal s_falling_edge is set (violet at 16,500 ns). This signal is sampled by the 100 MHz clock and when it is set, the s_puls signal is reset to 0. Due to the reset of s_puls, the s_en_fedge signal is reset at 17,000 ns. At this point the start_1MHz signal is also reset, which generated the 1 MHz start pulse to start or stop the debugging process. The s_falling_edge signal is reset at 17,500 ns, due to the disabled s_en_fedge signal.

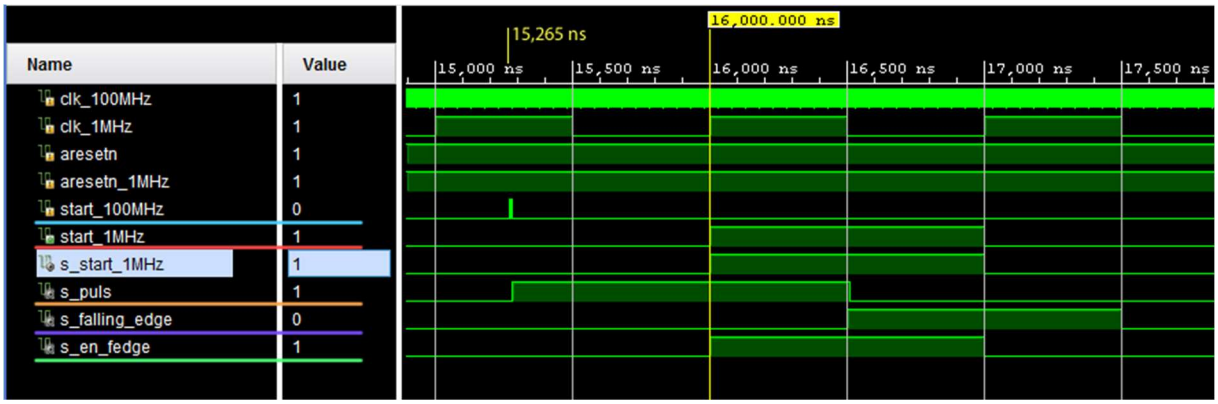


FIGURE 30: SIMULATION OF THE UNPKG_UDP_START_PULS BLOCK TO CREATE THE 1 MHZ START PULSE

4.8 IP CORE PKG_SAMPLES

The PKG_Samples block generates an AXIS data stream for the transfer into the RAM. The data stream contains the package type, the sample number and the four signal values. Each of these signals and the send_enable signal are routed from the DebugCoreModule block to the PKG_Samples block. To build the data stream, the send_enable signal has to be set. To build the AXIS data stream, the signals are converted from the 1 MHz clock domain into the 100 MHz clock domain. The frame size determines

[29] S_en_fedge stands for signal enable falling edge

the length of the AXIS data stream. The initTF[30] signal initializes the data transfer with the AXI DataMover into the RAM. Figure 31 shows the PKG_Samples block. The signal framesize defines the length of the AXIS data stream. The signal package_type is set to 1, which is defined as data package with the sampled signal data (Signal0, Signal1, Signal2 and Signal3) and the sample number (number_of_samples). The send_enable signal enables the build of the AXIS data stream. The AXIS data width can be adjusted in the IP settings. Currently the width is set to 32 bits. An overview of the inputs and outputs of this IP core is shown in Table 21 and Table 22 in the Appendix.

To lower the longest path in the design, pipeline stages were added between the DebugCoreModule and the PKG_Samples block. The pipeline stages are simple Flip-Flops to save the signal values. The pipeline stages are necessary to fulfill the timing constrains, which were generated from the FPGA design.

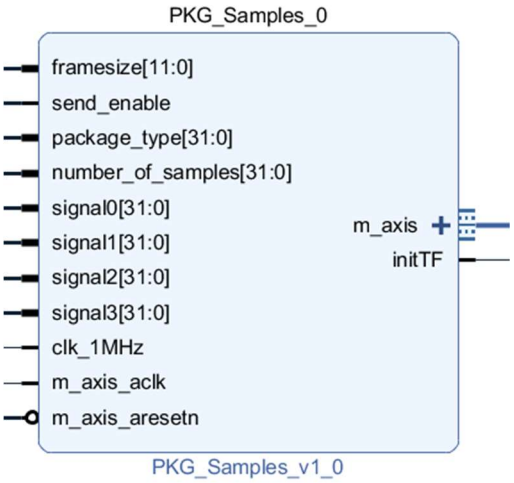


FIGURE 31: PKG_SAMPLES BLOCK WITH INPUTS AND OUTPUTS

Figure 32 shows the conversion of the 1 MHz send_enable signal (orange) to the 100 MHz initTF signal (red). The 100 MHz clock samples the 1 MHz clock and detects the rising edge of the 1 MHz clock. When send_enable is set and the rising edge was detected, the rising_1MHz signal is set. It gets delayed by one 100 MHz clock cycle (rising_1MHz_1). Both rising_1MHz signals (blue) generate the initTF signal for the 100 MHz clock cycle (20.000 ns). The falling edge of the 1 MHz clock resets the signals to detect the next rising edge.

This procedure makes it possible to convert the 1 MHz signal to a 100 MHz signal with the highest sample rate (1 MHz) and with lower sample rates. The send_enable signal is set with the sampling frequency. At the highest sample rate, the send_enable signal is constantly set to high, shown in Figure 32. At each rising edge of the 1 MHz clock, the initTF signal is generated. The send_enable signal is set with the sampling frequency.

[30] Init_TF stands for initialize transfer

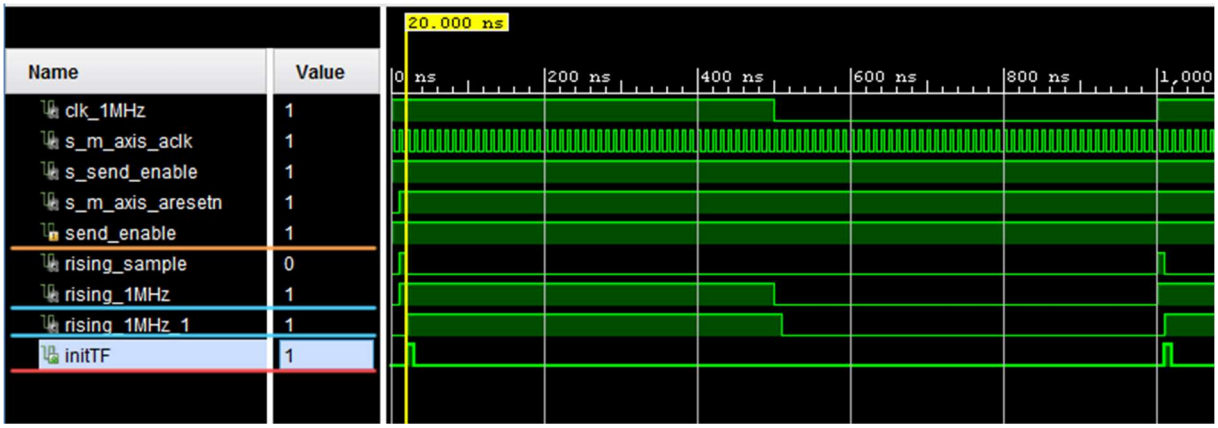


FIGURE 32: SIMULATION OF THE PKG_SAMPLES BLOCK TO GENERATE THE 100 MHZ INITTF SIGNAL

Figure 33 shows the packaged AXIS data stream (red). The data is sent with the AXIS master interface (red) to the S2MM_DataMover_Interface and forwarded to the AXI DataMover to write the data into the RAM. The initTF signal is generated from the send_enable signal (violet) and is set one clock cycle before the AXIS data transfer starts. The initTF signal is routed to the DataMoveCTL block, which initializes with the S2MM_DataMover_Interface the data transfer to the AXI DataMover and into the RAM. The data transfer starts at 2,020 ns. The last data word of the AXIS data stream is determined with the tlast signal (2,070 ns).

The created AXIS data stream contains the package type (1 for data package), the sample number (timestamp) and the 4 values of the selected signals for the debugging process (all orange).

FIGURE 33: SIMULATION OF BUILDING THE AXIS DATA STREAM WITH THE PKG_SAMPLES BLOCK

4.9 IP CORE DEBUGCOREMODULE

The DebugCoreModule was designed by me with MATLAB Simulink and provides the whole sampling functionality of the Advanced Inverter Debugger. The MATLAB model was converted to vhd files, which were included into a new Vivado project. Figure 34 shows the DebugCoreModule IP core. Most of the inputs are from the UNPKGModule to set up the debugging process. The Input_Signals is an interface with 300 signals, each 32 bits. 4 signals out of these 300 signals can be selected for the debugging process. The signal clk_1MHz_enable enables the module. All control signals, like c_NumOfSamples, c_SampleRate, c_CMD, c_TriggerTyp, c_TriggerValue, c_Signal0, c_Signal1, c_Signal2 and c_Signal3, are used to set up the debugging process with the adjusted parameters. The value of c_NumOfSamples

determines how long the Debug-Core is sampling. The value of c_SampleRate determines the sampling frequency. The value of c_CMD determines whether the debugging process is started or stopped/reset. It also determines whether a trigger is active. The value of c_TriggerType determines which trigger is active. The value of c_TriggerValue is the trigger value. The values of c_Signal0, c_Signal1, c_Signal2 and c_Signal3 determines the signals, which are selected for the debugging process. The signals debugged_Sig0, debugged_Sig1, debugged_Sig2 and debugged_Sig3 are the signals with the sampled values. They are used to build the data package. The signal debugged_NoS contains the sample number and is also used to build the data package. The sample number works as time stamp. The out_start_pkg1MHz signal enables the PKG_Samples block to build the AXIS data stream and start the AXIS data transfer into the RAM. The Resetn_1MHz signal is used to reset the DataMoveCTL block. All other signals were used for debugging and are not used anymore. An overview of the inputs and outputs of this IP core is shown in Table 23 and Table 24 in the Appendix.

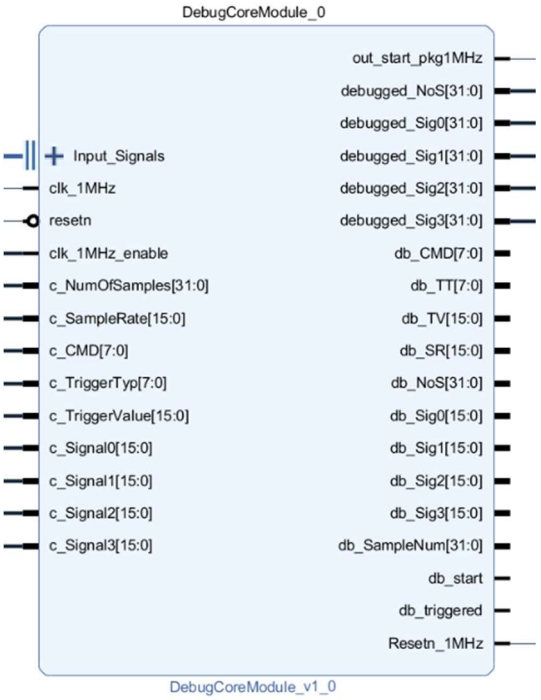


FIGURE 34: DEBUGCOREMODULE WITH INPUTS AND OUTPUTS

The DebugCoreModule is structured into several submodules shown in Figure 36.

The submodule Split_CMD_Bits splits the 8-Bit command signal into the different 1-bit signals. These signals are CMD_StartSampling, which start the debugging process, CMD_Reset, which stops/resets the debugging process, CMD_trigger, which enables the trigger and CMD_PreTrigger, which enables the Pre-Trigger[31]. The post-trigger[32] is active, when the trigger is enabled and the CMD_PreTrigger is not set.

The submodule Start_Control starts the sampling process. It also resets the sampling process.

The submodule Trigger_Control handles the whole triggering. It checks which trigger was selected and when to start the sampling depending on the present trigger settings.

The submodule Sample_Rate_Counter handles the sampling rate. Depending on the selected sample rate, the internal counter is changed to achieve the proper sample rate.

[31] Pre-Trigger, see chapter 4.12 Submodule Trigger_Control
[32] Post-Trigger, see chapter 4.12 Submodule Trigger_Control

The submodule Send_Control handles the sampling until the number of samples is reached or a reset occurs. It also resets the submodules, Sample_Rate_Counter, Start_Control and RingBuffer, when the adjusted number of samples is reached.

The submodule RingBuffer saves 100 signal values before the trigger event happens (Pre-Trigger). When the post-trigger is active, the signals are only routed through this block.

The submodule Signal_Selection contains the 300 to 1 multiplexer. There are 4 of these submodules in use due to the possibility to select 4 different signals for the debugging process. These 4 Signal_Selection blocks use most of the resources on the FPGA, which are used for the Advanced Inverter Debugger.

Figure 35 shows the simulation of the DebugCoreModule with an active post-trigger. The post trigger is the trigger that immediately activates the debugging process, when the trigger condition is met. The orange marked signals are the control signals, to set up the debugging process. At 2 us, all signals are updated to the values from the AXIS data stream. The c_CMD signal[33] has the value 0xA0, which sets up the debugging process with an active post-trigger. One clock cycle later at 3 us, the start debugging bit is reset again. The debugging process starts, when the value of the trigger signal (signal0) is less than the trigger value. The trigger value is 0. At 9 us, the trigger signal (debugged_Sig0, red) switches to the value -1, which activates the trigger. At this point, the sample counter (debugged_NoS, blue) starts increasing and also the signal to enable the AXIS data transfer is set (out_start_Pkg1MHz, violet). The selected signals for the debugging process are, signal0, which changes its value each clock cycle, signal2, signal3 and signal5, which are constants with the values 2, 3 and 5.

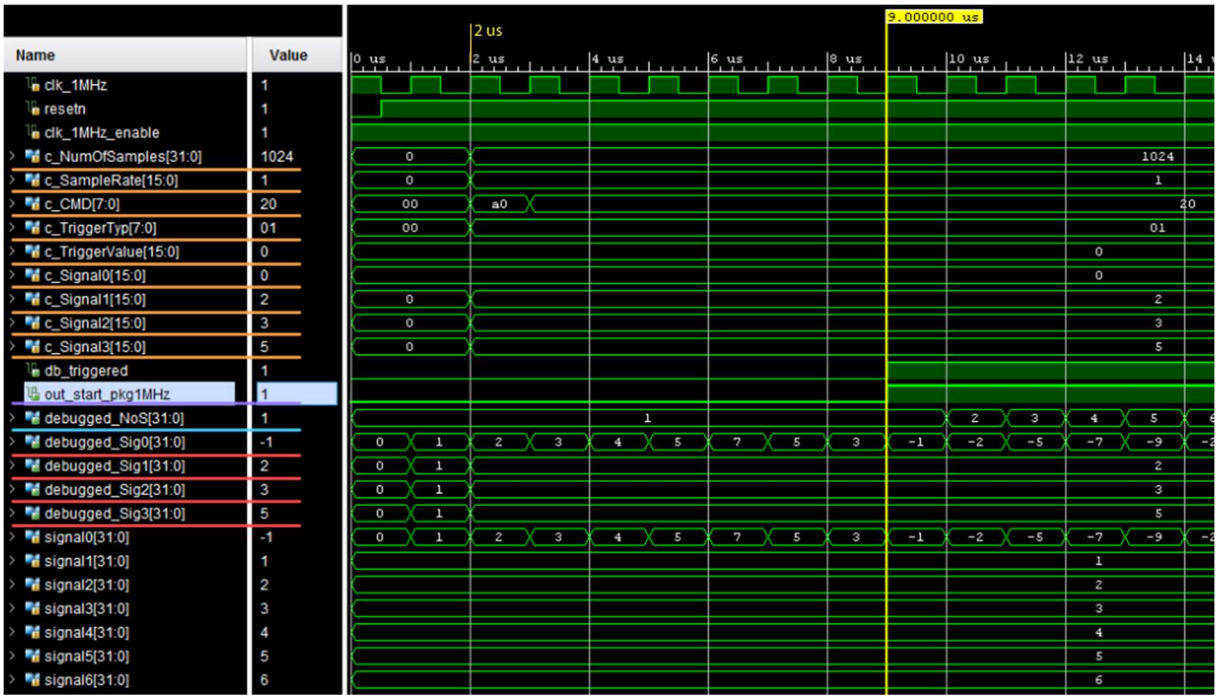


FIGURE 35: SIMULATION OF THE DEBUGCOREMODULE BLOCK WITH AN ACTIVE POST-TRIGGER

[33] CMD signal, see chapter 4.10 Submodule Split_CMD_Bits

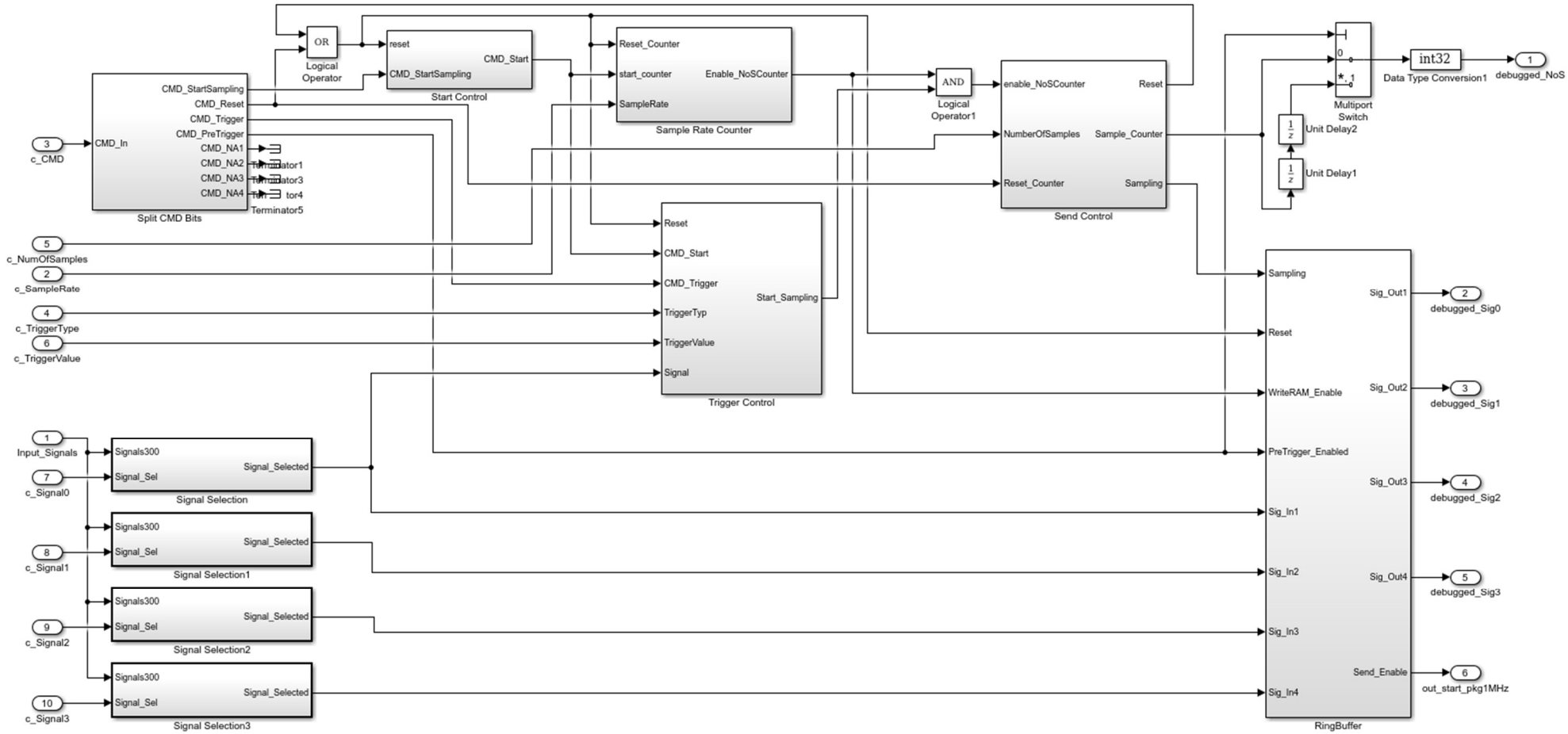


FIGURE 36: DEBUGCOREMODULE BLOCK OVERVIEW IN MATLAB SIMULINK WITH SUBMODULES

Figure 37 shows the simulation of the DebugCoreModule with an active pre-trigger. With the pre-trigger, up to 100 signal values of the 4 selected signals can be stored in ring buffers, before the trigger is activated. When no trigger occurs, the ring buffer overflows (cyclic overwrite). At 2 us, all signals are updated to the values from the AXIS data stream. The c_CMD signal has the value 0xB0, which sets up the debugging process with an active pre-trigger. At this point, the WriteRAM_Enable signal (blue) is set, which enables the write operation for the ring buffer. The selected signal values are written into the ring buffer. The selected signals are signal0 (also trigger signal), signal2, signal3 and signal5 (all marked blue). At 9 us, the trigger signal value is less than the trigger value, the trigger event occurs and the db_triggered signal (violet) is set. Due to the read operations from the ring buffers, the out_start_pkg1MHz signal (red) is set 2 clock cycles delayed, which initializes the AXIS data transfer. Also at 11 us, the sample number (debugged_NoS) increases and the signal values are read from the ring buffer (green). The read values start with 2, 3, 4 and 5 from the debugged_Sig0. The other 3 debugged signals (green) are constants with the values 2,3 and 5.

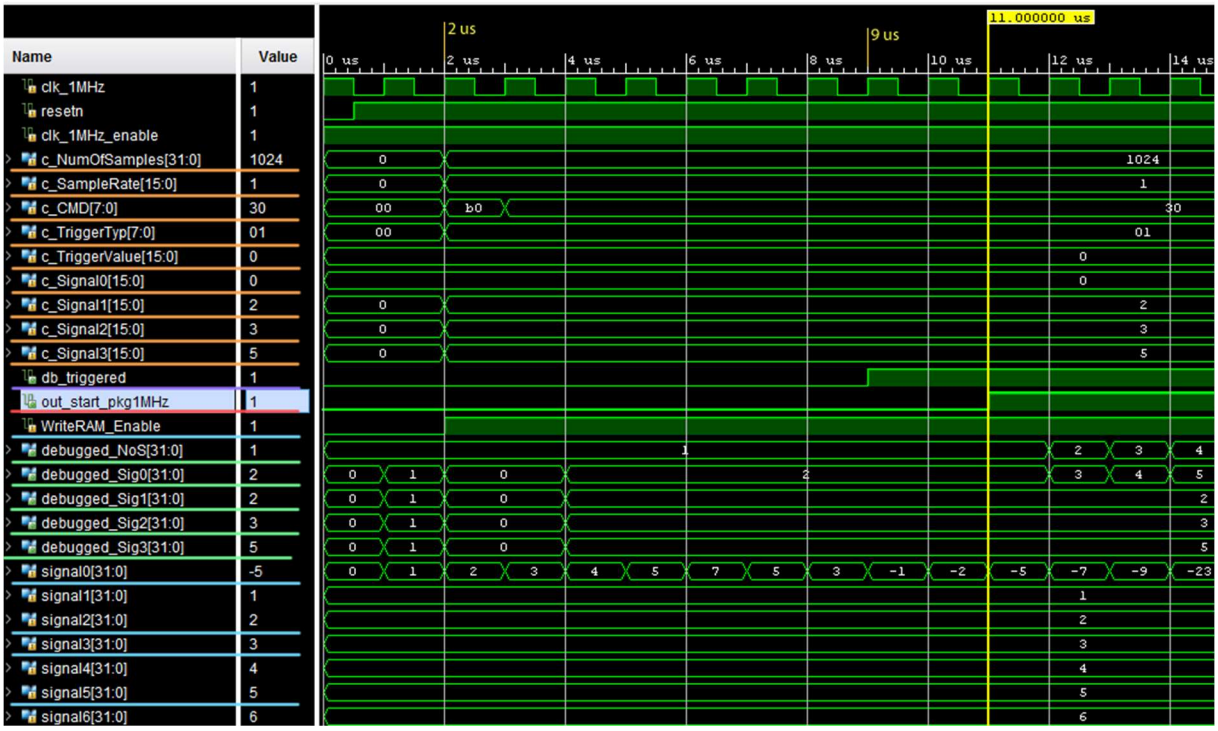


FIGURE 37: SIMULATION OF THE DEBUGCOREMODULE BLOCK WITH AN ACTIVE PRE-TRIGGER

4.10 SUBMODULE SPLIT_CMD_BITS

The submodule Split_CMD_Bits splits the 8-bit command signal into 1 bit signals. The block is shown in Figure 38. An overview of the inputs and outputs of this block is shown in Table 25 and Table 26 in the Appendix.

The CMD_In signal is shown in Figure 39. The 4 LSB are not accessed and can be used to extend the AID in the future. The start bit, CMD_StartSampling, is used to initialize the debugging process with a 1 MHz pulse. The reset bit, CMD_Reset, is used to stop the debugging process. The trigger bit, CMD_Trigger, enables the post-trigger. The pre-trigger is enabled, when both, the trigger and the Pre-Tr (CMD_PreTrigger) bits are set. When the start and reset bits are set during an unexpected error case, the reset signal is prioritized.

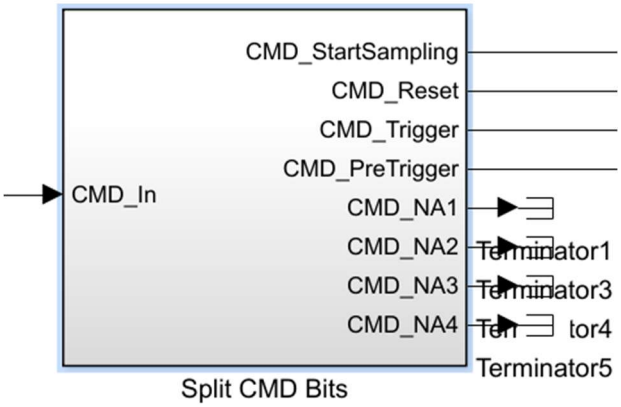


FIGURE 38: SPLIT_CMD_BITS BLOCK WITH INPUTS AND OUTPUTS

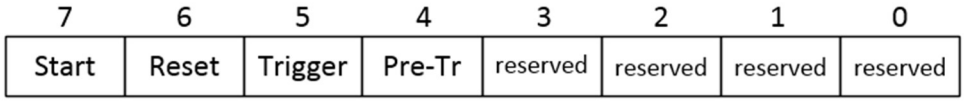


FIGURE 39: COMMAND BYTE WITH BIT DESCRIPTION

4.11 SUBMODULE START_CONTROL

The Start_Control block gets the 1 MHz CMD_StartSampling pulse from the Split_CMD_Bits block to generate a constant CMD_Start signal. This signal is set until a reset occurs. This reset can be the internal one, when the number of samples is reached or the CMD_Reset from the user-interface. When both input signals are set, the CMD_Start signal is not set and the debugging process is not started. Figure 40 shows the Start_Control block. An overview of the inputs and outputs of this block is shown in Table 27 and Table 28 in the Appendix.

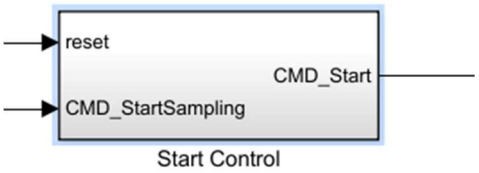


FIGURE 40: START_CONTROL BLOCK WITH INPUTS AND OUTPUTS

The Figures 41 and 42 show the simulation of the start and reset of the debugging process. To start the debugging process, the CMD_StartSampling signal sets the CMD_Start signal. It is constant high until a reset occurs. The reset can either be a stop debugging command (CMD_Reset) or an internal reset when the number of samples is reached to stop the sampling process. The CMD_start signal is set at 2 us in Figure 41 and is reset at 1,037 us in Figure 42, when the reset_1 signal is set.

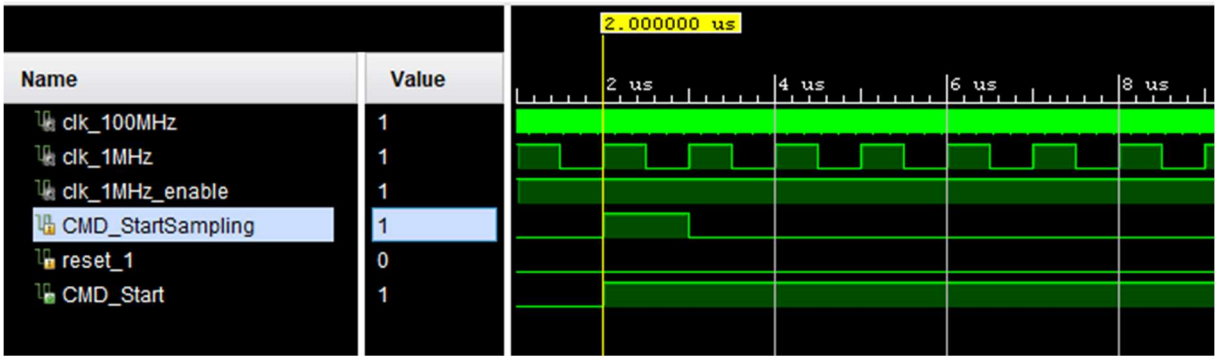


FIGURE 41: START_CONTROL SIMULATION OF THE CMD_START SIGNAL

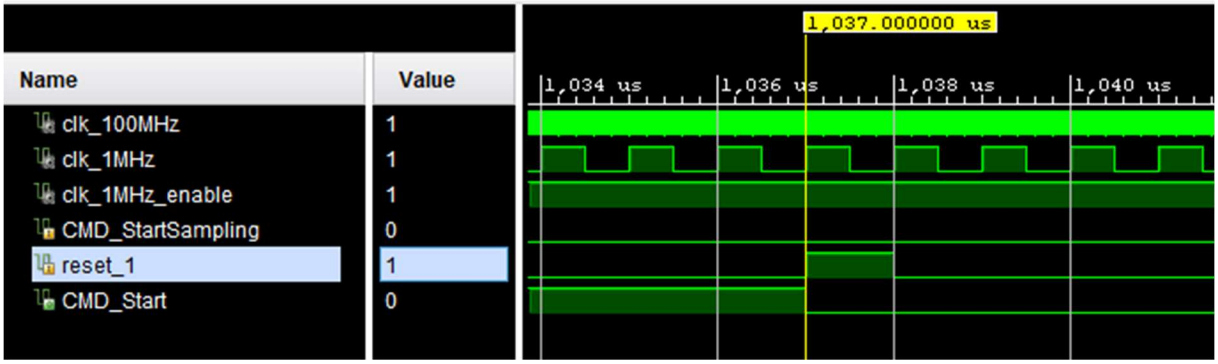


FIGURE 42: START_CONTROL SIMULATION OF THE CMD_START SIGNAL RESET

Figure 43 shows the simulation of the Start_Control block, when the signals to start and stop the debugging process are set at the same time. At 2 us, the signals CMD_StartSampling and reset_1 are set. The reset gets prioritized and the debugging process is reset. Due to the start and stop/reset package types, this case should not happen.

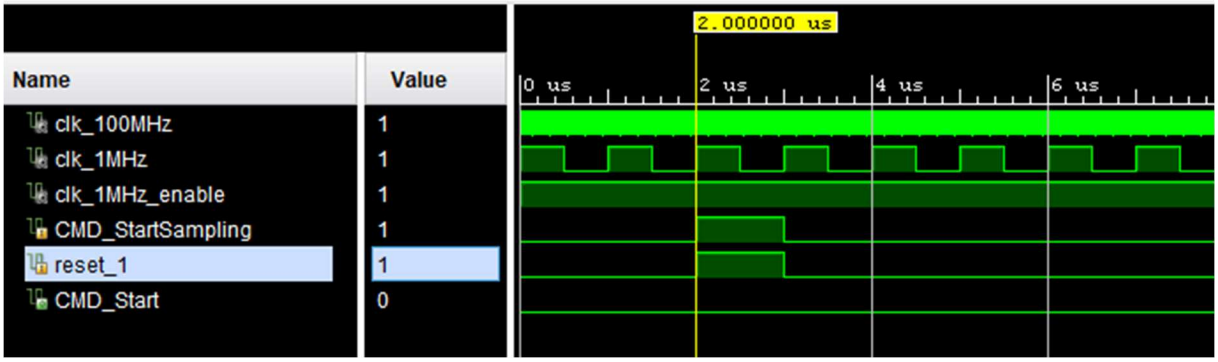


FIGURE 43: START_CONTROL SIMULATION WHEN BOTH INPUT SIGNALS ARE SET

4.12 SUBMODULE TRIGGER_CONTROL

The submodule Trigger_Control handles the trigger process. The submodule is shown in Figure 44. The internal structure is shown in Figure 45. Depending on the command signal, Pre- or Post-Trigger or even none trigger is selected. Depending on the TriggerType and the TriggerValue, the trigger point is changed. There are 3 Post-Triggers and 3 Pre-Triggers available. The signal and the trigger value are

converted into int32 signals. This makes a comparison possible, to check for the different trigger events. The trigger event can happen when the signal value is higher, lower or equal to the trigger value. Figure 46 shows an overview of the Trigger Selection block. If CMD_Trigger is low, no trigger is active and the Start_Sampling signal is set immediately. The Trigger_Control block handles only Post-Triggers. The Pre-Trigger is handled in combination with the RingBuffer block. The CMD_PreTrigger signal enables in combination with the Enable_NoSCounter signal the write operations into the ring buffer. A maximum of 100 data points can be stored (with cyclic overwrite) before the trigger event starts reading those values (enabled by Start_Sampling) to build the AXIS data stream. An overview of the inputs and outputs of this block is shown in Table 29 and Table 30 in the Appendix.

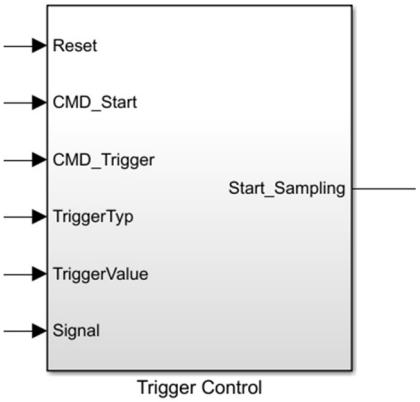


FIGURE 44: TRIGGER_CONTROL WITH INPUTS AND OUTPUTS

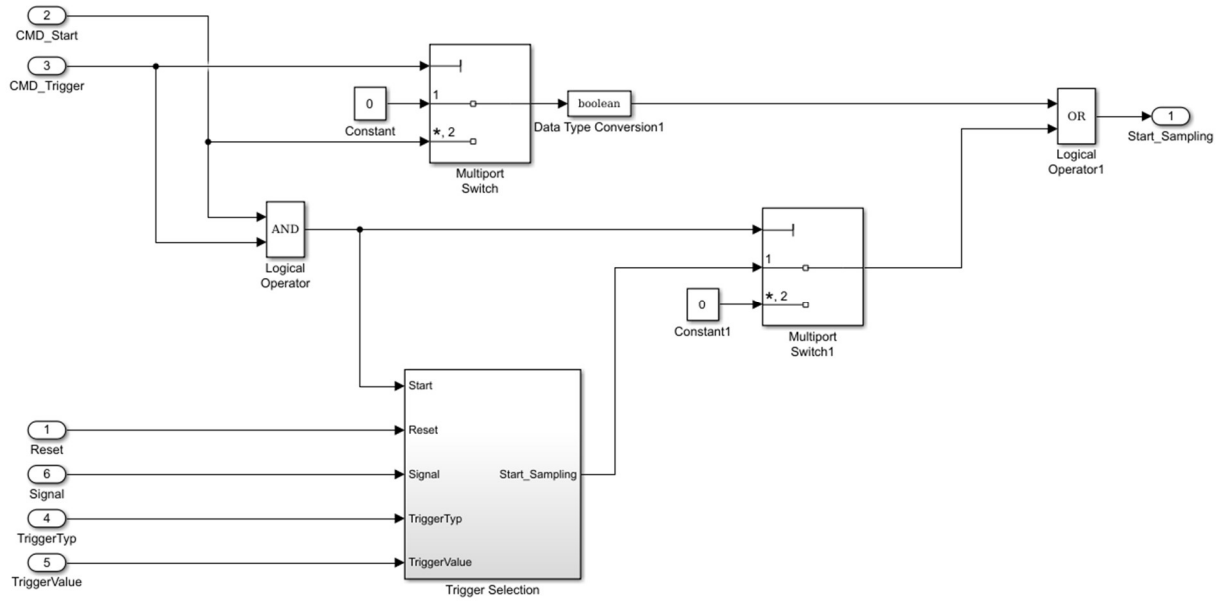


FIGURE 45: INTERNAL STRUCTURE OF THE TRIGGER_CONRTOL MODULE

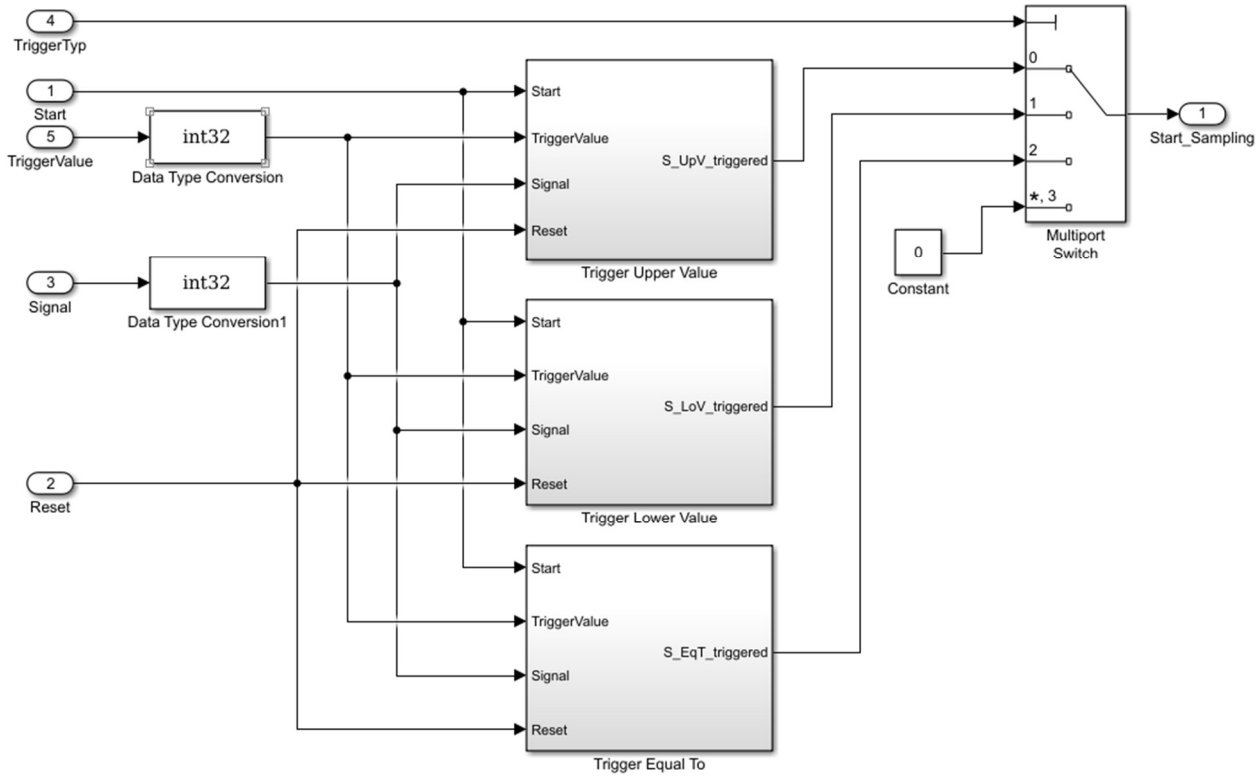


FIGURE 46: INTERNAL STRUCTURE OF THE TRIGGER_SELECTION MODULE

4.12.1 TRIGGER TYPE DESCRIPTION

The functionality of the Pre- and Post-Trigger of the Trigger_Control block is the same. The Post-Trigger is handled with this module but the Pre-Trigger is handled in combination with the ring buffer. The CMD_PreTrigger enables with the Enable_NoSCounter signal the write operations into the ring buffer. When the trigger event happens, the read operations start. The Trigger_Control block handles the different trigger types, like higher, less, or equal to a value, shown in Table 1. Each of this options can be selected for the Post- and Pre-Trigger.

TABLE 1: TRIGGER TYPE DESCRIPTION

Trigger	TriggerType Value	Description
Signal value is higher than the trigger value	0	The trigger activates the sampling, when the signal value is higher than the trigger value. If the Pre-Trigger is active, the read operations from the ring buffer is also enabled.
Signal value is lower than the trigger value	1	The trigger activates the sampling, when the signal value is lower than the trigger value. If the Pre-Trigger is active, the read operations from the ring buffer is also enabled.
Signal value is equal to the trigger value	2	The trigger activates the sampling when the signal value is equal to the trigger value. If the Pre-Trigger is active, the read operations from the ring buffer is also enabled.

4.12.2 FUNCTIONALITY OF THE TRIGGER

Figure 47 gives an overview of the trigger process. The signal, which is used for the trigger, is compared with the trigger value. This is done with the operating frequency of the AID, 1MHz. When the selected sample rate is 1 MHz, the trigger event starts the sampling process immediately. Figure 47 shows the trigger process with a lower sampling rate. Here, the signal value crosses the trigger value shortly after the sample point. The sampling process starts with the next sample point. The same principle is also used for the other trigger types.

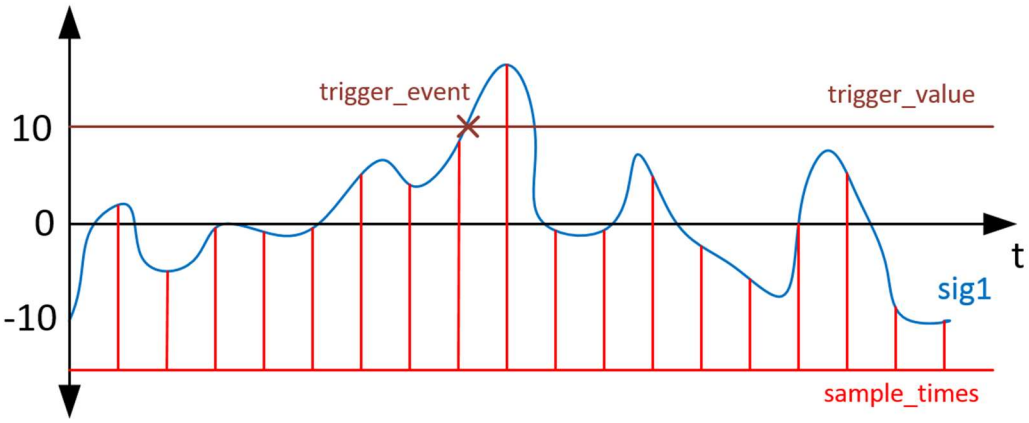


FIGURE 47: TRIGGER FUNCTIONALITY

Figure 48 shows the simulation of the trigger process. The signal value (Signal_rsvd, red) changes every clock cycle and can be a positive or negative integer. At 2 us, the trigger is enabled with the command signals CMD_Start and CMD_Trigger and set up with the trigger type and the trigger value (marked orange). The trigger value is set to 0 and the trigger type is set to 1, which means the trigger event happens, if the signal value is below the trigger value. At 9 us, the signal S_LoV_triggered (lower than value) is enabled. Also the other signals, S_UpV_triggered (above value) and S_EqT_triggered (equal to) can be enabled at this point. With the trigger type, the correct trigger event is selected and the Start_Sampling signal (blue) is set.

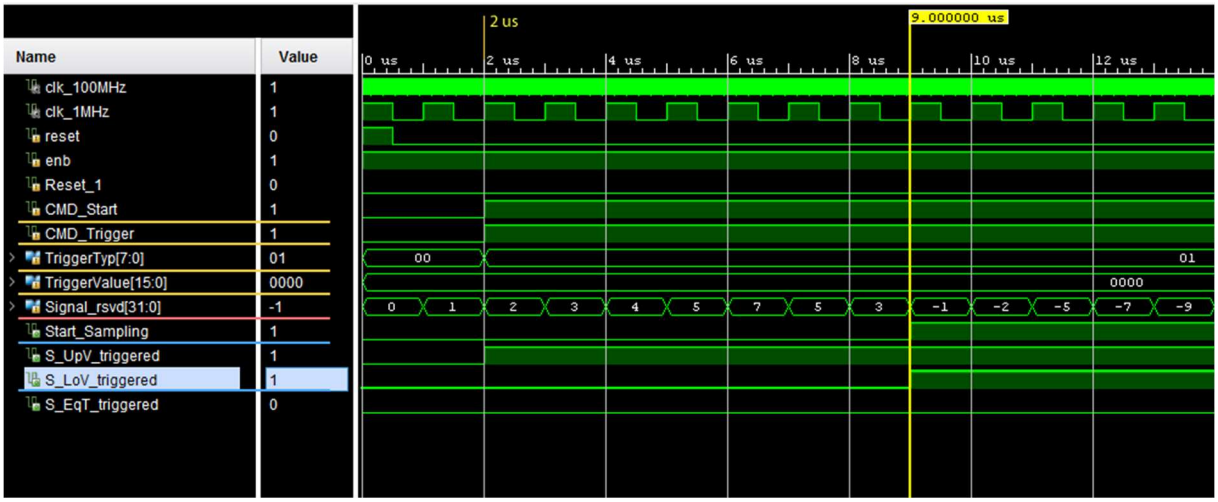


FIGURE 48: TRIGGER_CONTROL SIMULATION WITH SIGNAL VALUE LOWER THAN TRIGGER VALUE

Figure 49 shows the simulation when no trigger is active. The CMD_Trigger signal (blue) is not set and therefore no trigger is active. The CMD_Start signal sets the Start_Sampling signal (red) to start the debugging process immediately.

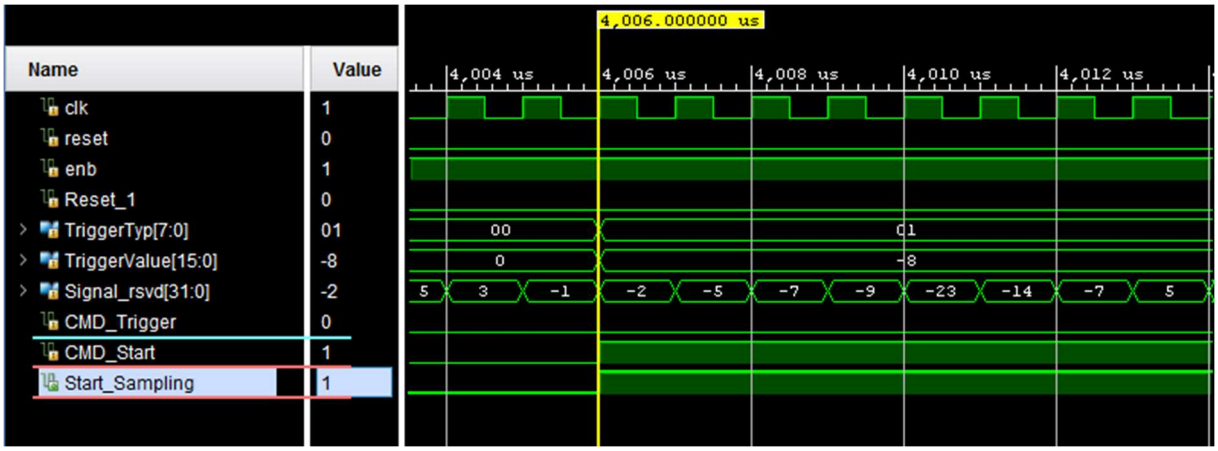


FIGURE 49: TRIGGER_CONTROL SIMULATION WHEN NO TRIGGER IS ACTIVE

4.13 SUBMODULE SEND_CONTROL

The submodule Send_Control handles the sampling process. It counts the numbers of samples and if the adjusted NumberOfSamples is reached, the internal Reset is set. This Reset resets the Send_Control block, the Trigger_Control block, the Sample_Rate_Counter block, the RingBuffer and the internal sample counter of the Send_Control block. The internal sample counter increases when the Start_Sampling signal from the Trigger_Control block and the Enable_NoSCounter from the Sample_Rate_Counter are set. The counter increments, when both signals are set. This is necessary due to different sample rates. Figure 50 shows the Send_Control block. Figure 51 gives a more detailed overview of this block. The Sample_Counter is the timestamp for the sampled signal values. With the Sampling signal, the data transfer into the RAM is initialized and when the Pre-Trigger is active, the read operation from the ring buffer is initialized with the data transfer into the RAM. The Reset_Counter signal resets the counter, when the stop debugging command is sent from the user-interface. An overview of the inputs and outputs of this block is shown in Table 31 and Table 32 in the Appendix.

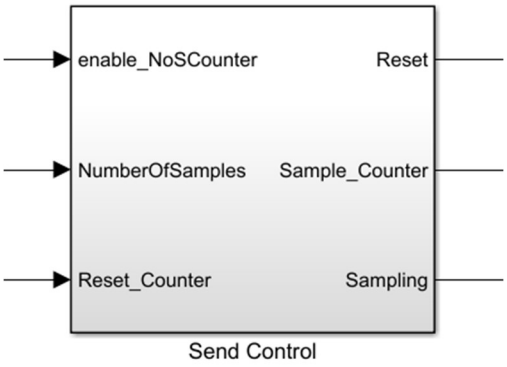


FIGURE 50: SEND_CONTROL WITH INPUTS AND OUTPUTS

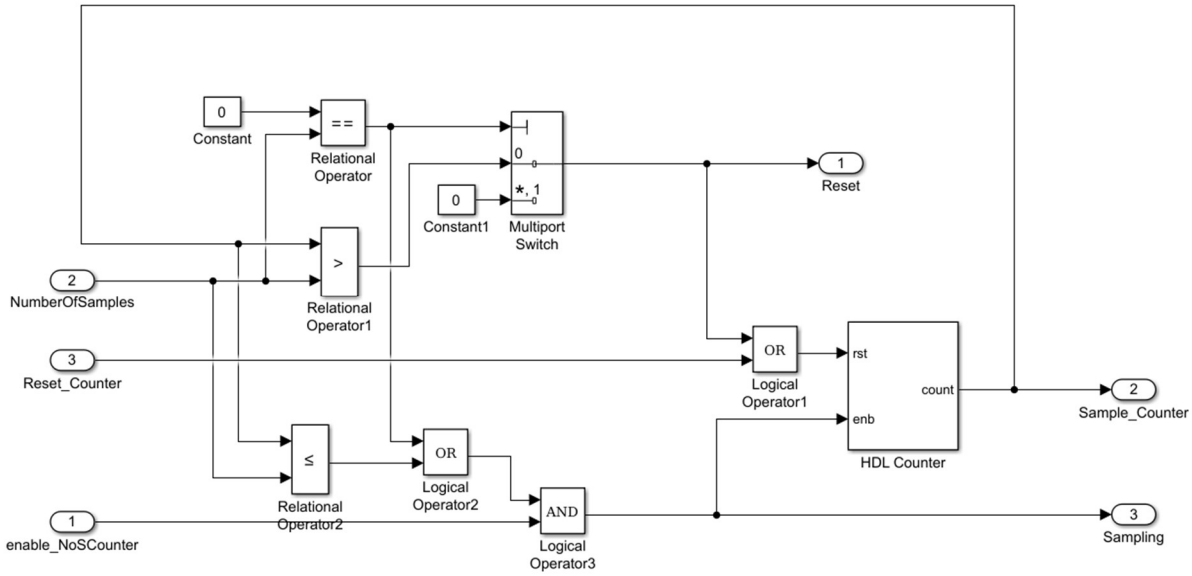


FIGURE 51: SEND_CONTROL MODULE OVERVIEW

The Send_Control block gets the selected number of samples, which can be between 1024 and 999424. The sample counter is compared to the selected number of samples after each increase. The Reset signal is set, when the counter value is higher. It resets the internal sample counter and the other submodules. It is also possible to select 0 numbers of samples, which means the sampling process runs to infinity until the CMD_Reset stops it. It also resets the sample counter and the other submodules. The CMD_Reset is done with the user-interface.

Figure 52 shows the simulation of the Send_Control block. The selected sampling frequency is 250 kHz. Therefore, the sample counter increases with each fourth clock cycle. When the sample counter reaches the number of samples (blue), which is 1024, then the counter is reset and the other submodules are also reset.

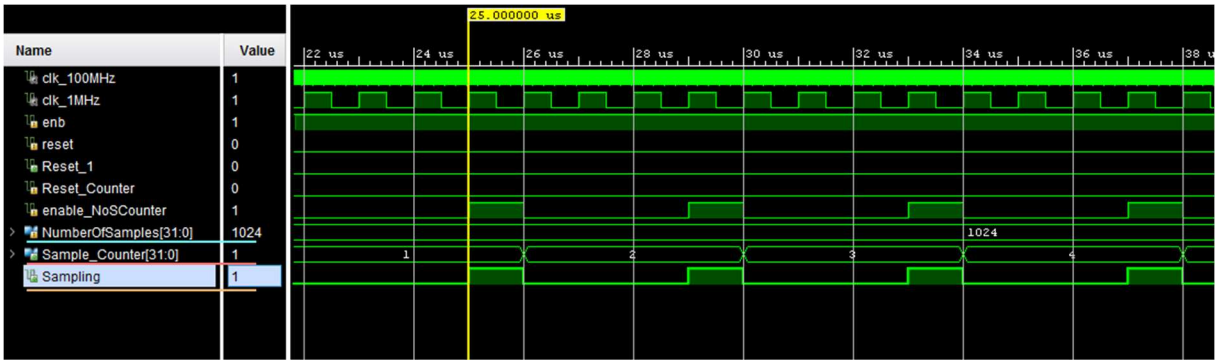


FIGURE 52: SEND_CONTROL SIMULATION WITH A SAMPLE RATE OF 250 KHZ

Figure 53 shows the simulation of the Send_Control block when the adjusted number of samples is reached. The sample frequency is 1 MHz, therefore the Sampling signal (orange) is constantly set and the sample counter (red) is increased with each clock cycle. The number of samples (blue) is set to 1024 and at 1,048 ns, the sample counter reaches 1024 samples. Therefore, the sample counter is reset. The reset is also used, to reset the other submodules with the signal Reset_1.

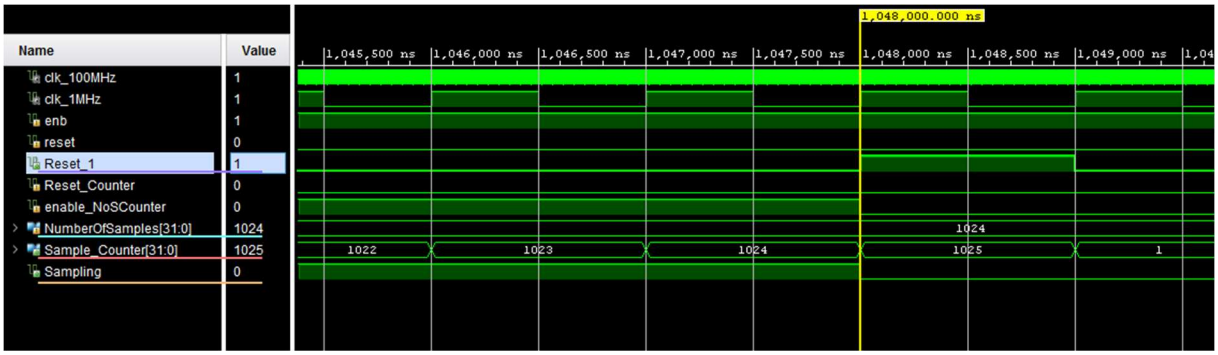


FIGURE 53: SEND_CONTROL SIMULATION WHEN THE NUMBER OF SAMPLES IS REACHED

4.14 SUBMODULE SAMPLE_RATE_COUNTER

The submodule Sample_Rate_Counter handles the sample frequency. The value of the SampleRate signal determines the sample frequency. It is the compare value to the internal counter. The start_counter signal enables the counting process to determine the sample frequency. Each time, the counter reaches the value of the SampleRate signal, the counter is reset and the Enable_NoSCounter is set. The Reset_Counter signal resets the Sample_Rate_Counter block. The block is shown in Figure 54. The internal structure of the Sample_Rate_Counter block is shown in Figure 55. An overview of the inputs and outputs of this block is shown in Table 33 and Table 34 in the Appendix.

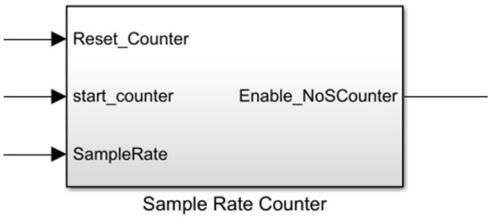


FIGURE 54: SAMPLE_RATE_COUNTER WITH INPUTS AND OUTPUTS

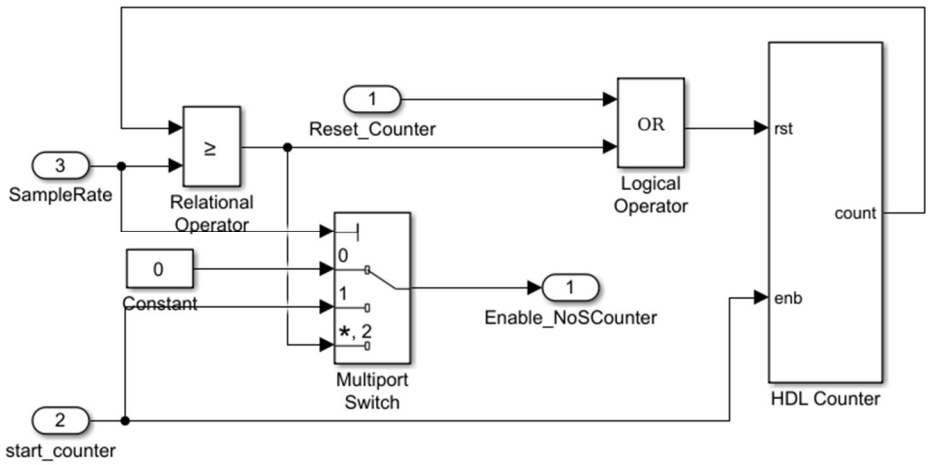


FIGURE 55: INTERNAL STRUCTURE OF THE SAMPLE_RATE_COUNTER MODULE

Table 2 shows the different sample frequencies, which can be selected with the user-interface. The selectable sample frequencies are between 1 kHz and 1 MHz. To start the debugging process with the selected frequency, the internal counter must be compared to the sample rate value. This value is

calculated with the maximum frequency (1 MHz) and the selected sample frequency, shown in Equation 1. When the sample frequency is set to 1 MHz, the internal counter is reset each clock cycle. Also the Enable_NoSCounter signal is set each clock cycle. If other frequencies are selected, the internal counter increases with a frequency of 1 MHz until the sample rate value is reached and enables the Enable_NoSCounter signal. The counter is reset again and starts increasing until the start_counter signal is reset.

$$SampleRate\ [value] = \frac{maxFrequency[Hz]}{SampleRateFrequency\ [Hz]}$$

TABLE 2: SELECTABLE SAMPLE FREQUENCIES FOR THE DEBUGGING PROCESS

SampleRate	Sample Frequency
[value]	[kHz]
1	1000
2	500
4	250
5	200
8	125
10	100
16	62.5
20	50
25	40
32	31.25
40	25
50	20
64	15.625
80	12.5
100	10
125	8
160	6.25
200	5
250	4
320	3.125
400	2.5
500	2
625	1.6
800	1.25
1000	1

4.15 SUBMODULE RINGBUFFER

The Submodule RingBuffer handles the Pre-Trigger setting. The module is shown in Figure 56. When the Pre-Trigger is set, 100 signal values can be saved into Block-RAMs before the actual trigger event happens. This gives more information about the behavior of the signals before an event was received. As BRAMs, Simple Dual Port RAMs of the MATLAB Simulink library are used.

The WriteRAM_Enable signal enables the RingBuffer to write the signal values into the BRAMs. The RingBuffer will cyclic overwrite after 100 entries. When the trigger event occurs, the entries are read from the BRAMs. If the Pre-Trigger is disabled, the signal data is routed through the RingBuffer module. Otherwise the signal data from the RingBuffer is used.

The Sampling signal sets the Send_Enable signal. It also activates the read operation by increasing the read address for the BRAMs. When the Pre-Trigger is active, the Send_Enable signal is delayed by 2 clock cycles due to the write and read operations of the BRAMs. A more detailed overview of the RingBuffer module is shown in Figure 57.

The Reset signal resets the internal counters. The write and read counter are used as memory addresses for the BRAMs. An overview of the inputs and outputs of this block is shown in Table 35 and Table 36 in the Appendix.

The whole RingBuffer submodule is structured into the submodules RingBufferCTL and RingBufferSig. The RingBufferCTL contains the counters to address the memory. The RingBufferSig contains the BRAM blocks and the logic to select the right signal data for the output ports.

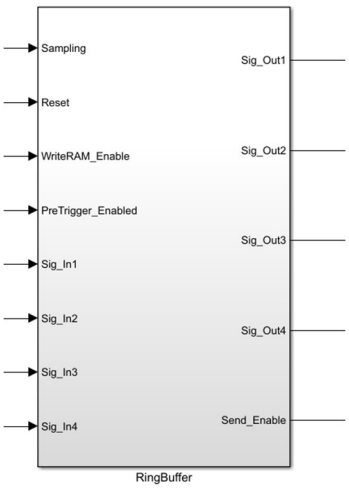


FIGURE 56: RINGBUFFER WITH INPUTS AND OUTPUTS

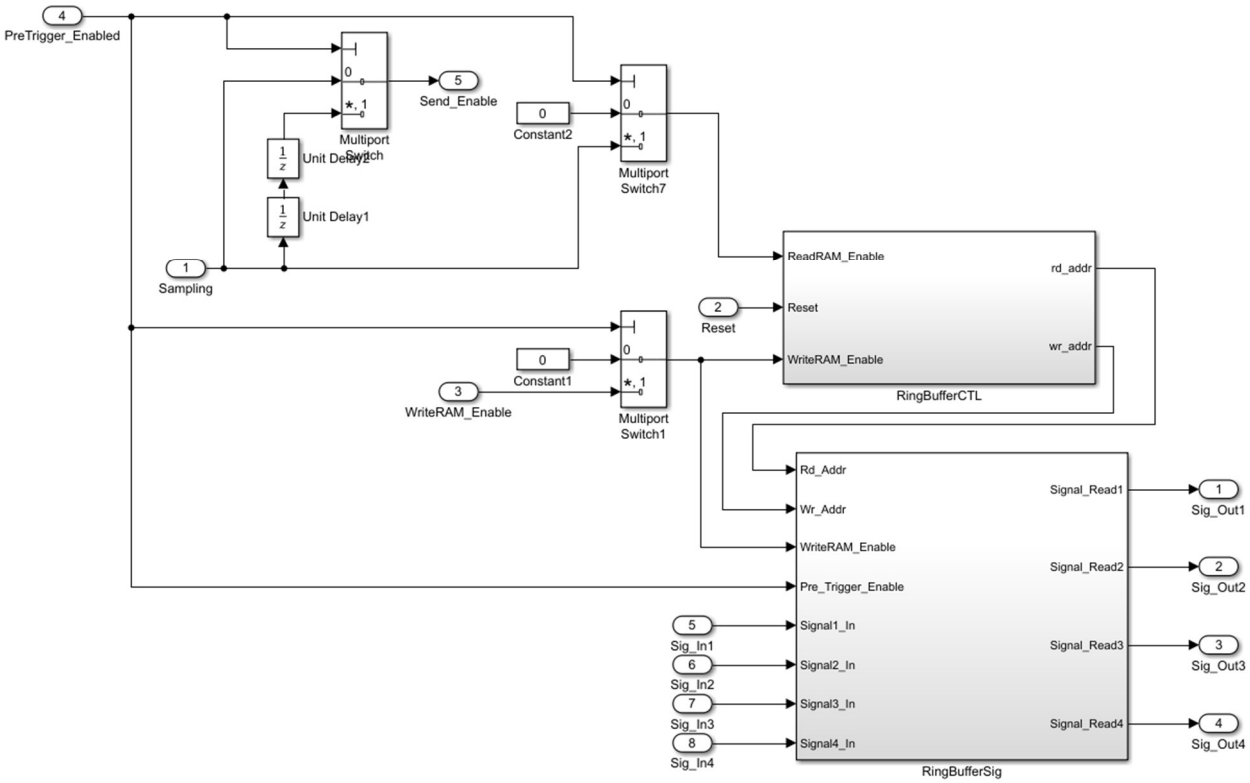


FIGURE 57: INTERNAL STRUCTURE OF THE RINGBUFFER MODULE

The submodule RingBufferCTL contains the counters for the write and read operations. The ReadRAM_Enable signal increments the read counter and the WriteRAM_Enable signal increments the write counter. When they reach the value 102, they overflow and start from 0 again. The counter values

are used as BRAM addresses (rd_addr and wr_addr). When the trigger event occurs at the first sample, a write and read operation of the BRAMs are necessary. To avoid concurrent access, the read address is delayed by 1 clock cycle. This makes sure, that the signal data is written into the BRAMs before they are read. When the trigger event occurs, the ReadRAM_Enable signal is set and the read counter starts to increase. The read counter moves always with the same gap to the write counter. Figure 58 shows the internal structure of the RingBufferCTL module. An overview of the inputs and outputs of this block is shown in Table 37 and Table 38 in the Appendix.

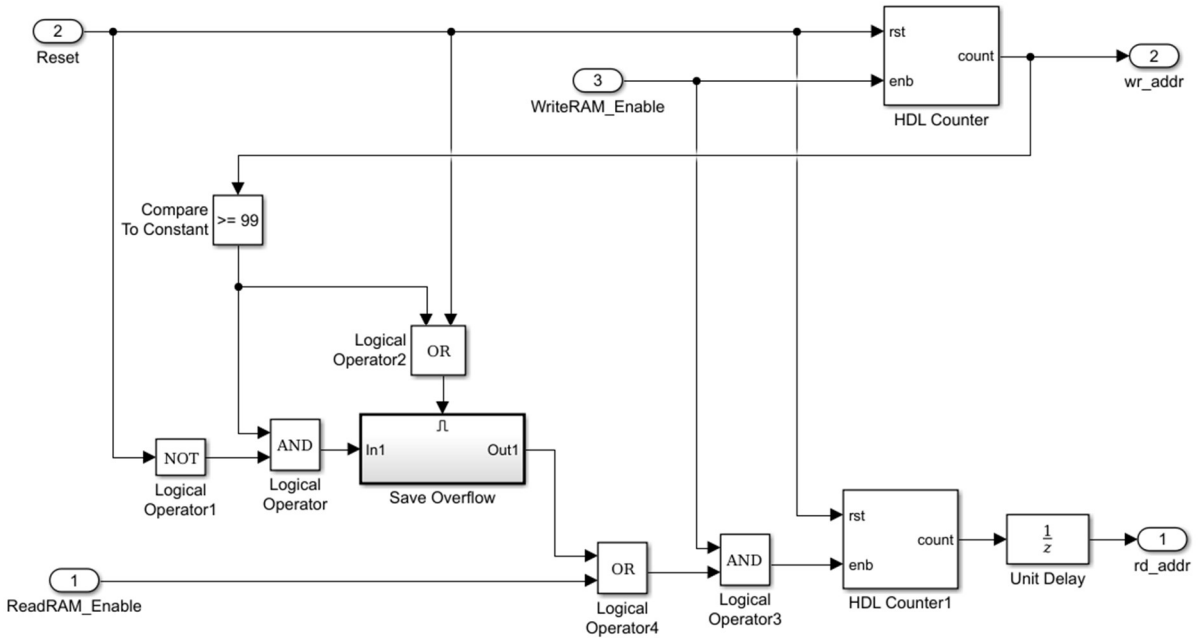


FIGURE 58: INTERNAL STRUCTURE OF THE RINGBUFFERCTL MODULE

The submodule RingBufferSig checks if the Pre-Trigger is active and which data will be routed to the output ports. If the Pre-Trigger is active, the saved signal values from the BRAMs are selected, otherwise the 4 signals are directly routed through the RingBufferSig block. The read and write address for the BRAMs are controlled with the RingBufferCTL block. With the WriteRAM_Enable signal, new values are written into the BRAMs and the previous values are read. Due to this behavior of the Simple Dual Block RAM, there is no need for an extra read signal. Every debugging signal has its own ring buffer, shown in Figure 59. An overview of the inputs and outputs of this block is shown in Table 39 and Table 40 in the Appendix.

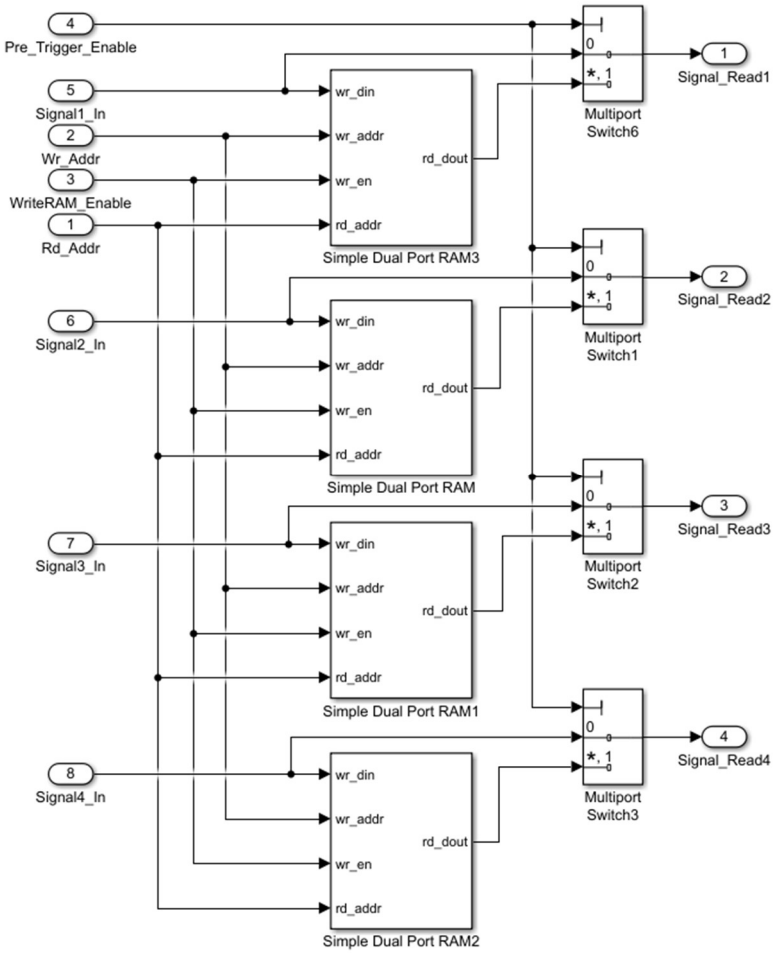


FIGURE 59: INTERNAL STRUCTURE OF THE RINGBUFFERSIG MODULE

Figure 60 shows the read and write pointer to address the BRAMs of the ring buffer. When the CMD_Start signal is set, the sample rate counter starts to work (0). The write address increases with each enable signal from the sample rate counter block (2). This is only done, when the pre-trigger is enabled. For every sample point, the data is written into the BRAMs, until the pre-trigger event occurs. If there is no trigger event, the read address starts incrementing, when the write address reaches 99 (3). The read pointer follows the write pointer with a constant delay of a 100 entries (4). After reaching the value 102, the address starts from 0 again and the saved values will be overwritten.

Figure 61 shows the read and write pointer to address the BRAMs of the ring buffer, when the pre-trigger event occurs. First, both counters start at 0 (1). Then, the write address increases and the sample points are written into the BRAMs (2). When the write pointer reaches 9, the trigger event occurs and the saved samples are read from the read pointer location. The read pointer increases and follows the write pointer with a constant delay (3 and 4).

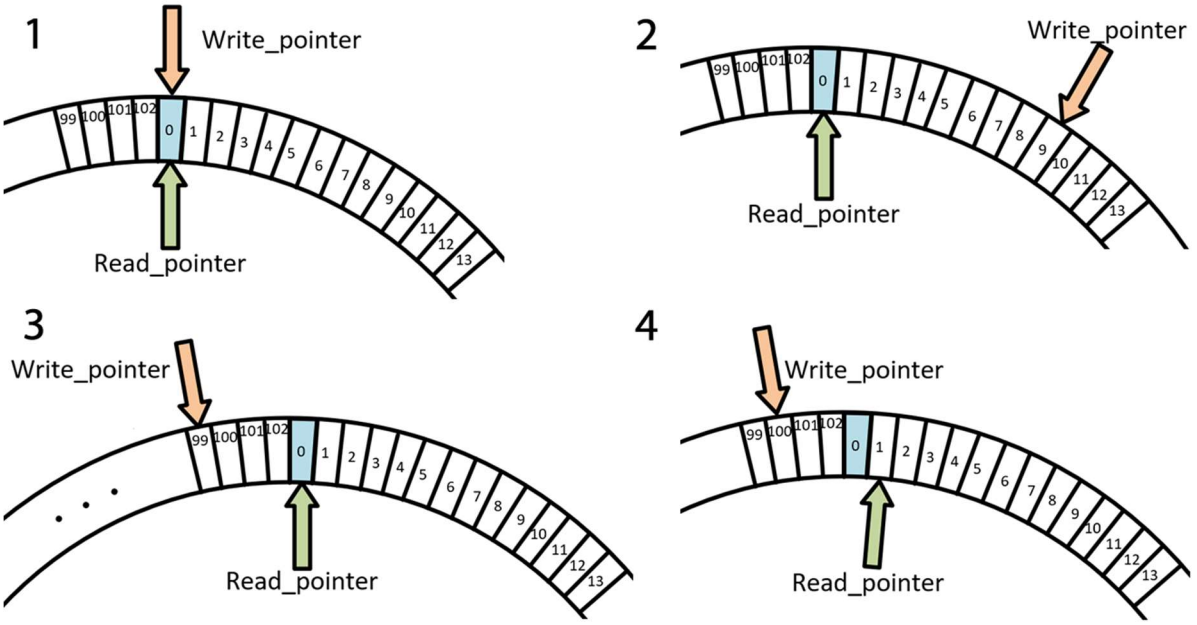


FIGURE 60: RING BUFFER WITH READ AND WRITE POINTER MOVEMENT WHEN NO TRIGGER EVENT OCCURS

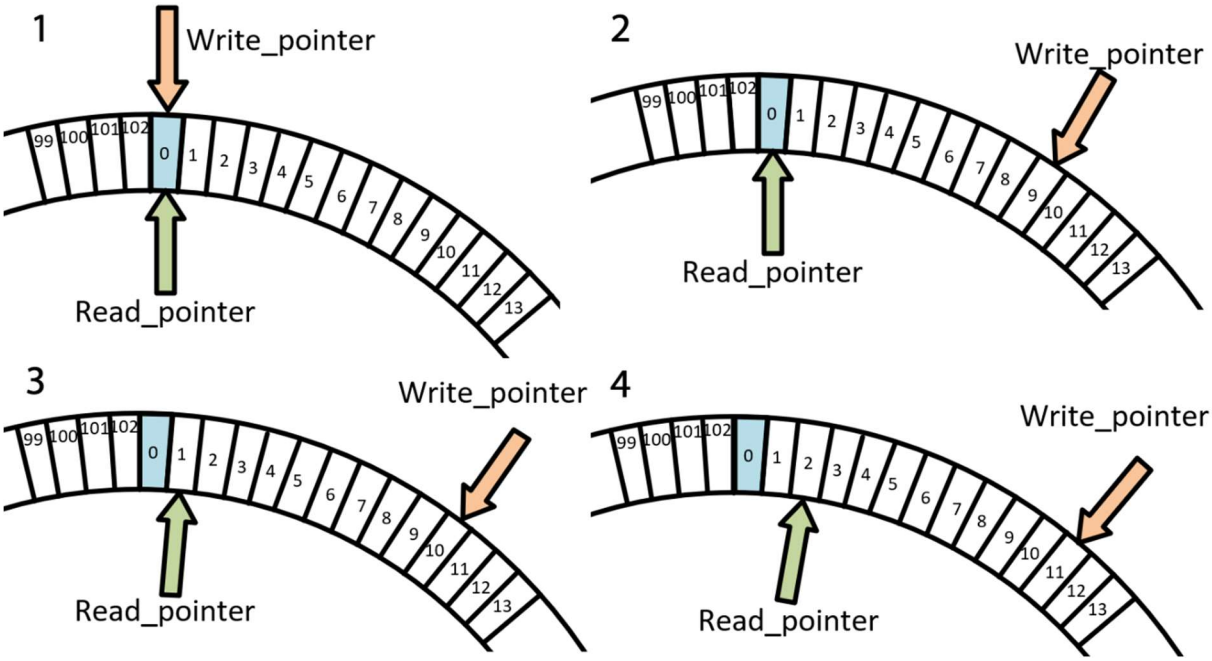


FIGURE 61: RING BUFFER WITH READ AND WRITE POINTER MOVEMENT WHEN A TRIGGER EVENT OCCURS

The Figure 62 shows the simulation of the RingBuffer block with an active Pre-Trigger. At 2 us, the signals WriteRAM_Enable and PreTrigger_Enabled (both orange) are set and the write operations start. The signal data is written into the BRAMs. The write address (Wr_Addr) begins at 0 and increases with the sample frequency, which is 1 MHz. At 6 us, the trigger event occurs, when the value of the signal Sig_In (red) is above 5 (trigger value) and the Sampling signal (violet) is set. This activates the read operations from the BRAMs with a delay of 1 clock cycle. At 8 us, the read operation is finished and the read values are routed to the output signals. The signal Sig_Out (red) has the value 2, which was written into the BRAMs at the first write operation. The read address (Rd_Addr) increments and the Send_Enable signal is set (both marked blue) to build the AXIS data stream. The read counter follows the write counter with a constant delay, depending on the trigger event.

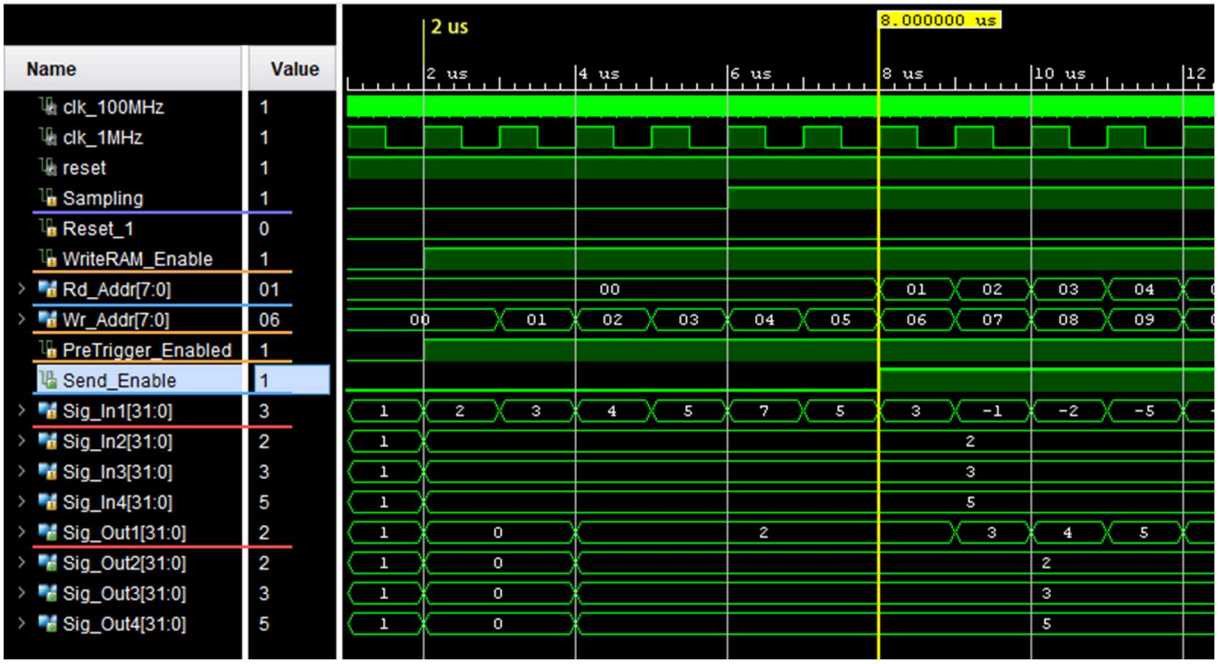


FIGURE 62: SIMULATION OF THE RINGBUFFER MODULE WITH AN ACTIVE PRE-TRIGGER

Figure 63 shows the simulation of the RingBuffer block, when a post-trigger is active. The PreTrigger_Enabled signal is 0 and therefore no addresses are incremented. The BRAMs are not used. At 6 us, the trigger event happens (Sig_In value is above 5) and the Sampling signal is set. Due to the active post-trigger, the Send_Enable signal is set immediately to build the AXIS data stream.

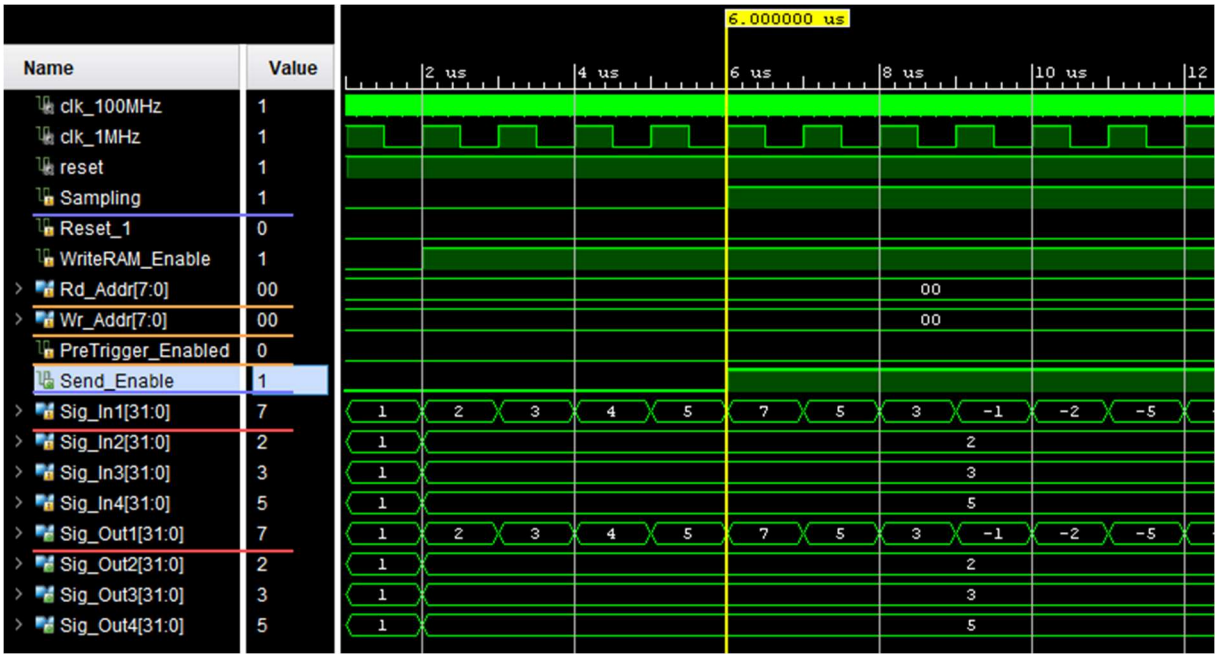


FIGURE 63: SIMULATION OF THE RINGBUFFER MODULE WITH AN ACTIVE POST-TRIGGER

4.16 SUBMODULE SIGNALSELECTION

The submodule SignalSelection is a 300 to 1 multiplexer. It selects 1 signal out of 300 input signals. The signal selection is done with the signal Signal_Sel. The value of this signal routes the selected signal to the output port. To address the 300 signals correctly, the DebugCoreModule has its own interface for the 300 input signals. This interface is called Input_Signals and is used by integrating the DebugCoreModule into the Xilinx Vivado Design. It contains all 300 input signals with 32 bit, each. The SignalSelection block is shown in Figure 64. The Advanced Inverter Debugger has 4 SignalSelection blocks, one block for each signal to debug. An overview of the inputs and outputs of this block is shown in Table 41 and Table 42 in the Appendix.

FIGURE 64: SIGNAL_SELECTION WITH INPUTS AND OUTPUTS

To avoid, that the 300 to 1 multiplexer is synthesized with the internal multiplexers of the FPGA, the idea was to build this big multiplexer with lookup tables. The SignalSelection block is structured into submodules. The 300 input signals are separated into 3 times 100 signals (Figure 65) and these 100 signals are again separated into 10 times 10 signals (Figure 66). The 10 signals are controlled by the SignalEnableTen block (Figure 67). It depends, which signal is selected (Figure 68) and the SignalEnableTen block enables the corresponding SignalSelectTen block and routes the selected signal to the output. When the enable signal is disabled, the output signal is set to 0 (Figure 69). All of the outputs are combined with bitwise OR operations to get the chosen signal. Due to this process, the 300 to 1 multiplexer is synthesized with lookup tables in the FPGA. This is also shown in the Utilization report of the design that no internal Mul7 and Mul8 are used.

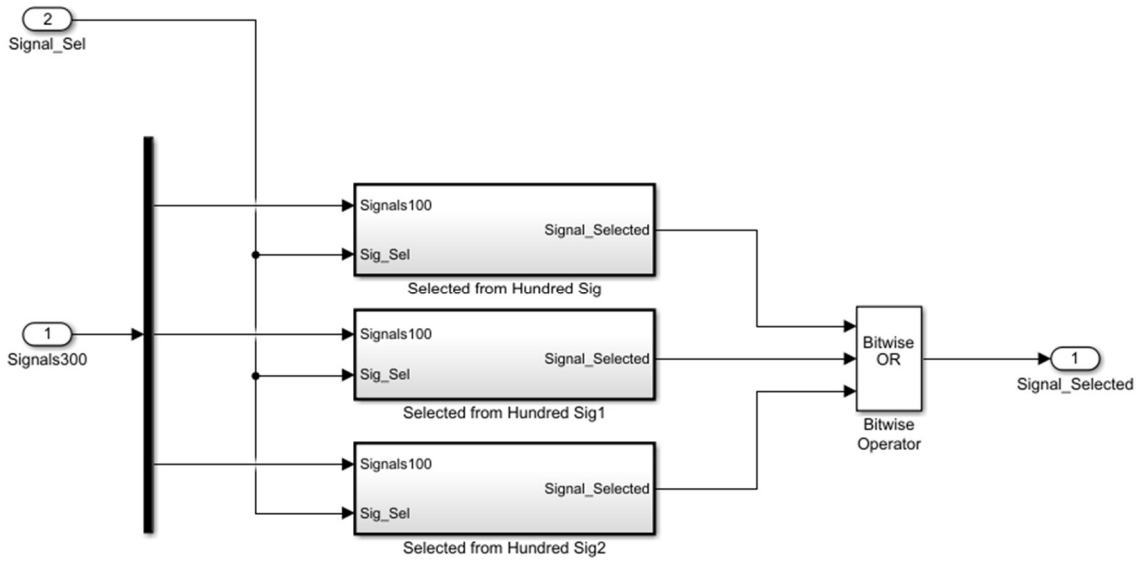


FIGURE 65: SIGNALSELECTION WITH 300 SIGNALS SEPARATED IN 3 TIMES 100 SIGNALS

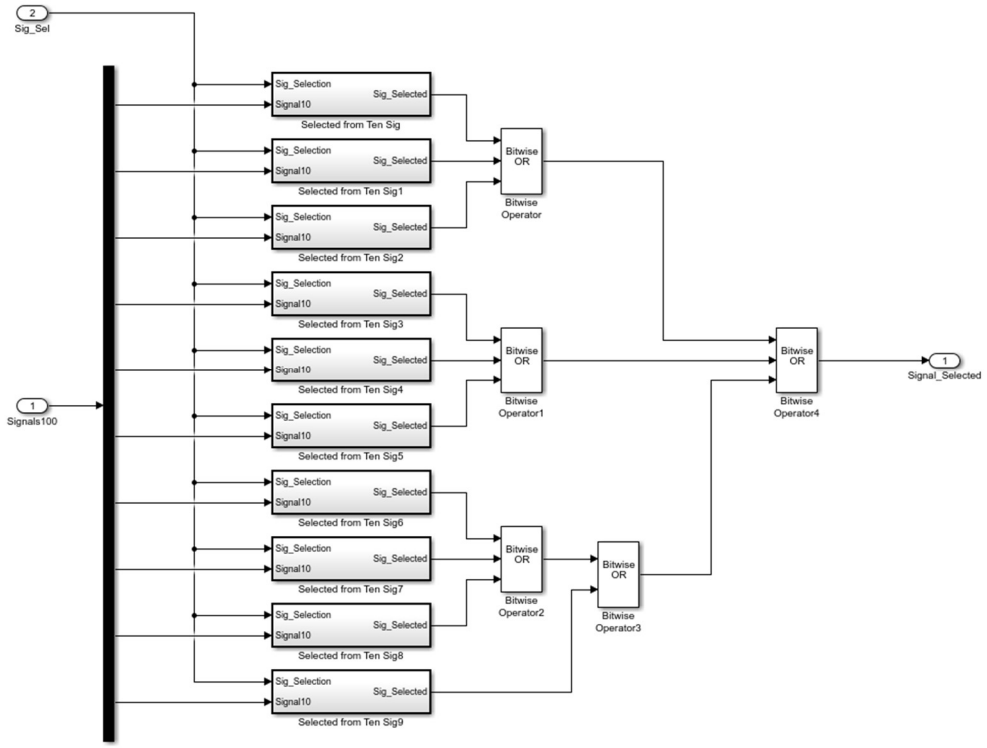


FIGURE 66: SIGNALSELECTION WITH 100 SIGNALS SEPARATED INTO 10 TIMES 10 SIGNALS

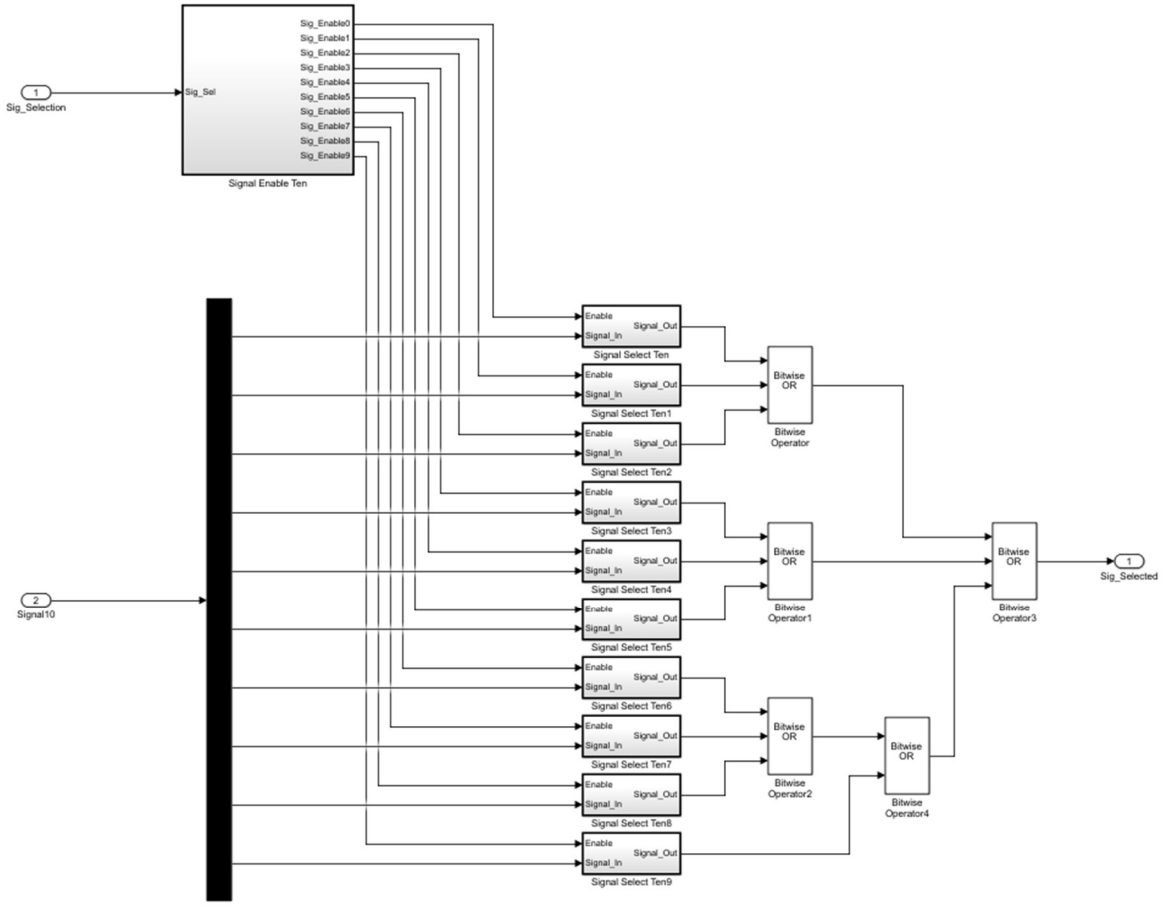


FIGURE 67: SIGNALSELECTION WITH AN ENABLE CONTROL AND 10 SIGNAL SELECTION BLOCKS

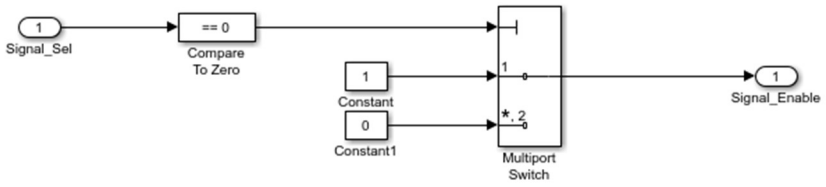


FIGURE 68: SIGNALSELECTION WITH THE CONTROL SIGNAL

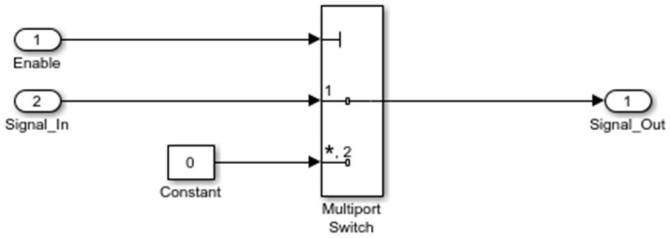


FIGURE 69: SIGNALSELECTION TO ENABLE THE SELECTED SIGNAL FOR THE DEBUGGING PROCESS

4.17 PIPELINES AND CLOCK DOMAIN CORSSING

The Pipeline IP cores are simple registers to reduce the longest path in the FPGA design. This is important to fulfill the timing constrains. The timing constrains were generated from the design. The pipeline block is shown in Figure 70. The signal width can be chosen in the IP settings of the block. This block is used for signals between the UNPKGModule and the DebugCoreModule. Pipelines are also used between the DebugCoreModule and the PKG_Samples block. An overview of the inputs and outputs of the Pipeline block is shown in Table 43 and Table 44 in the Appendix.

The pipeline block was modified with an input interface with 300 and 40 signals. These pipelines (Pipeline40 and Pipeline300) are used as input interfaces for the packaged AID40[34] and AID300[35]. Each input signal of the interfaces are 32 bits and can't be changed. An overview of the inputs and outputs of the Pipeline300 IP core is shown in Table 45 and Table 46 in the Appendix.

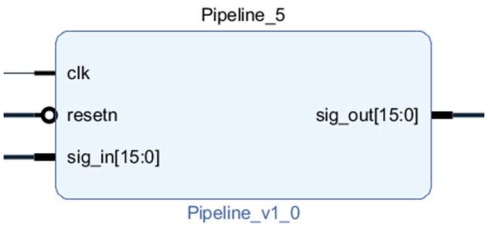


FIGURE 70: PIPELINE WITH INPUTS AND OUTPUTS

[34] AID40, Advanced Inverter Debugger IP core with 40 input signals
[35] AID300, Advanced Inverter Debugger IP core with 300 input signals

The ClkDomainCrossing block converts a signal from one clock domain to another clock domain. This block is used to convert the 1 MHz out_start_pkg1MHz signal from the DebugCoreModule to the 100 MHz send_enable signal for the PKG_Samples block. The block is shown in Figure 71. The width of the input and output signal can be set in the IP settings. An overview of the inputs and outputs of the ClockDomainCrossing IP core is shown in Table 47 and Table 48 in the Appendix.

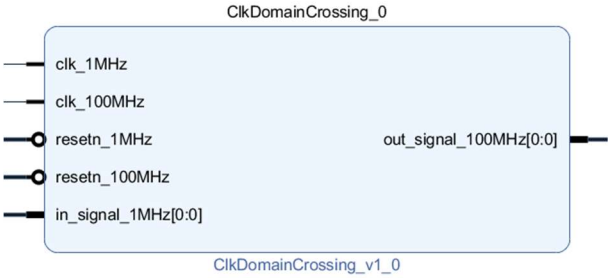


FIGURE 71: CLKDOMAINCROSSING IP CORE WITH INPUTS AND OUTPUTS

Figure 72 shows the structure of the clock domain crossing in general. The first Flip-Flop is used to synchronize the input signal to the rising edge of clk1. Then, the second Flip-Flop is used to synchronize the signal B to clk2. It is still possible, that the output C is metastable due to a violation of the setup or hold time. To avoid this, another Flip-Flop is added, to get valid data after the third Flip-Flop.

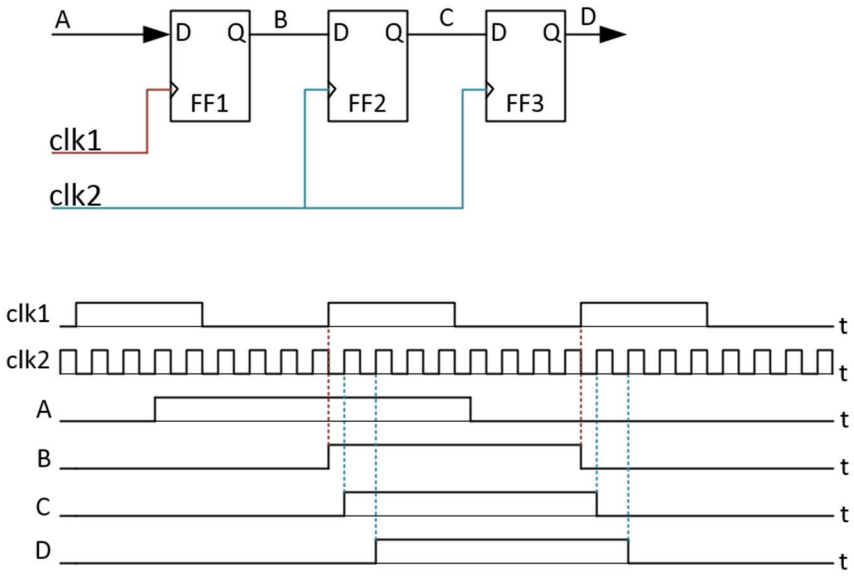


FIGURE 72: FUNCTIONALITY OF THE CLOCK DOMAIN CROSSING

Figure 73 shows the clock domain crossing simulation from the 1 MHz clock domain to the 100 MHz clock domain. It also shows the Flip-Flop stages for the clock domain crossing. At the rising edge of the 1 MHz clock, the stage1 is set to the in_signal_1MHz. At the rising edge of the 100 MHz clock, stage2 takes over the value of stage1. By the next rising edge, stage3 takes over the value of stage2. At 2,020 us, the signal out_signal_100MHz (red) is set to the value of stage3.

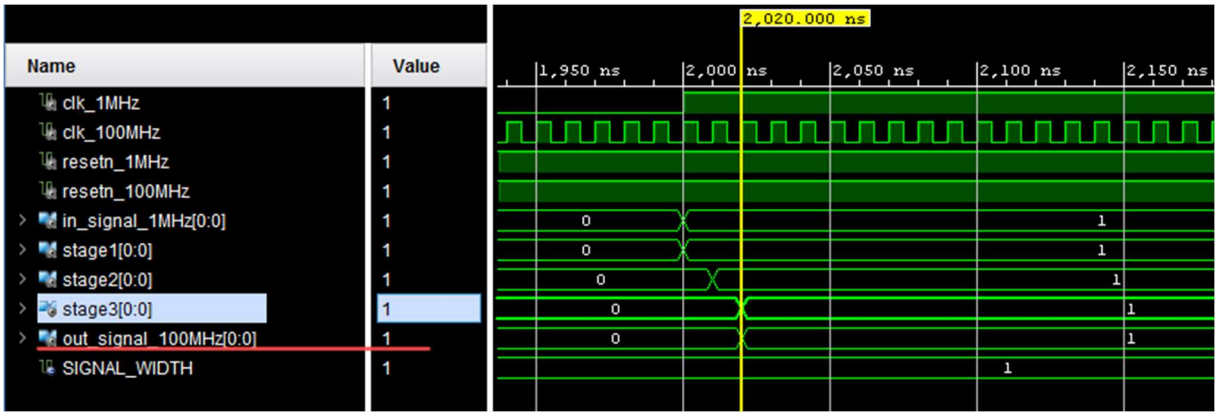


FIGURE 73: SIMULATION OF THE CLOCK DOMAIN CROSSING

4.18 SIGNAL GENERATOR IP CORE SIG_GEN300

The Sig_Gen300 module generates the test signals for the practical tests of the Advanced Inverter Debugger. It generates 300 signals. Each signal is 32 bits. The first 20 signals are generated as counters. The first counter starts counting from a negative value and counts upwards until the maximum is reached and starts at the minimum again. The starting points of all counters are -40, -30, -20, -10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140 and 150. Each counter increases the value by 1. Every counter has a different counting speed. The first one counts with 1 MHz, the second counter needs 2 clock cycles, the third counter needs 3 clock cycles … and the last counter needs 20 clock cycles to increase the counter value by 1. The maximum value is 50 higher than the starting point. When the maximum is reached, the counters start from the starting values again. The other 280 signals are just constants with the values 20 – 299. These values were chosen to test the signal selection. The Sig_Gen300 block is shown in Figure 74. The output signals are combined to an interface with the name Gen_Signals, which stands for generated signals. An overview of the inputs and outputs of the Sig_Gen300 IP core is shown in Table 49 and Table 50 in the Appendix. This signal generator is also available with 40 signals with the name Signal40Generator. The 40 signals contains 20 counters and 20 constant signals (values 20-39).

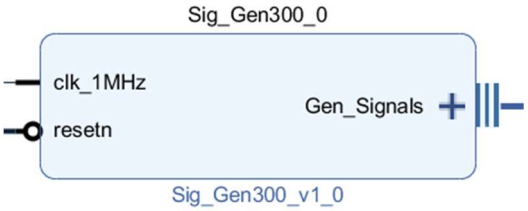


FIGURE 74: SIG_GEN300 INPUTS AND OUTPUTS

The Figure 75 shows the generation of the test signals for the Debug-Core. The first 20 signals are counters, which start at different values and increase with a different rate. At the signal 20, the signal values are constant with the signal value of the signal index.

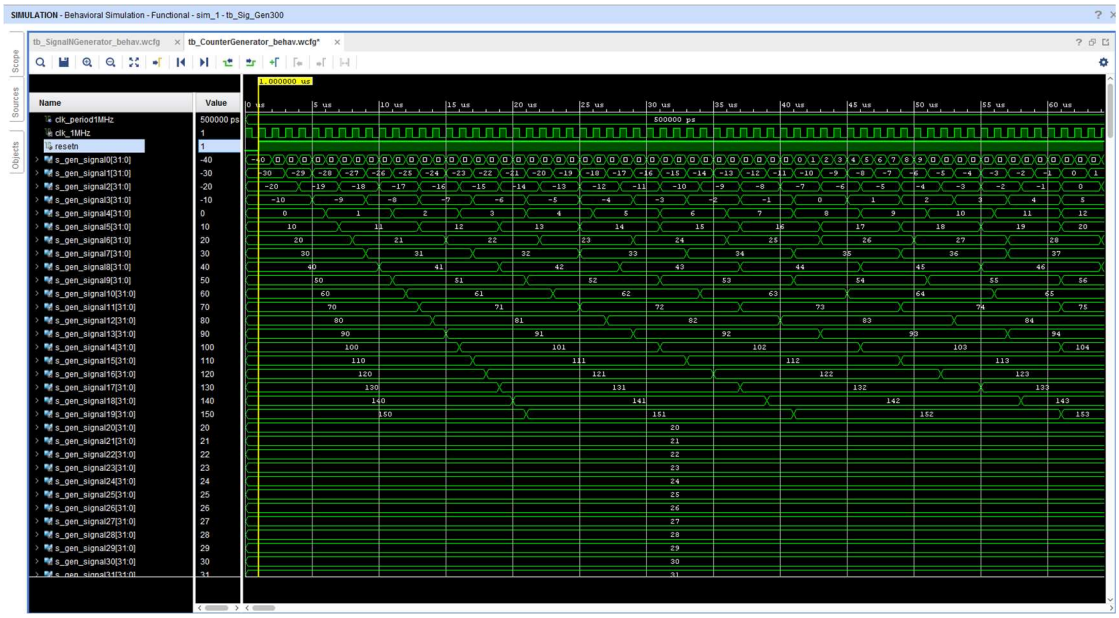


FIGURE 75: SIMULATION OF THE SIG_GEN300 IP CORE TO GENERATE THE 300 TEST SIGNALS

4.19 PACKAGED ADVANCED INVERTER DEBUGGER IP CORE

The packaged AID IP core combines the UNPKGModule, DebugCoreModule, DataMoveCTL, PKG_Samples, Pipelines and ClockDomainCrossing blocks to one IP core. The AID IP core is available with 40 and 300 input signals. The input signals are combined to an interface.

For the AID IP core with 300 input signals, the interface name is Input_Signals. This interface was created by me and is also used for the Sig_Gen300 IP core. For this interface, the Input_Signals.xml and the Input_Signals_rtl.xml files are used. The packaged AID IP core with 300 input signals is named AID300 and is used to test it on the ZedBoard.

For the AID IP core with 40 input signals, the interface name is debugSig. This interface was created by Stephan Hochmüller and is already used in the design on the Controller Board. For this interface, the debugSig.xml and the debugSig_rtl.xml files are needed. The packaged IP core with 40 input signals is named AID and is used to test it on the Controller Board.

To use the interfaces, the files must be copied into the IP repository, the interface names have to be selected under the point User. The Figure 76 shows both interfaces. The left interface (debug_Sig) is for 40 input signals and the interface, Input_Signals, is for 300 input signals. After selecting the right interface, the signals must be mapped. The mapping sets the physical connections, how the single signals are connected when the interface is used. The packaged AID IP core is shown in Figure 77.

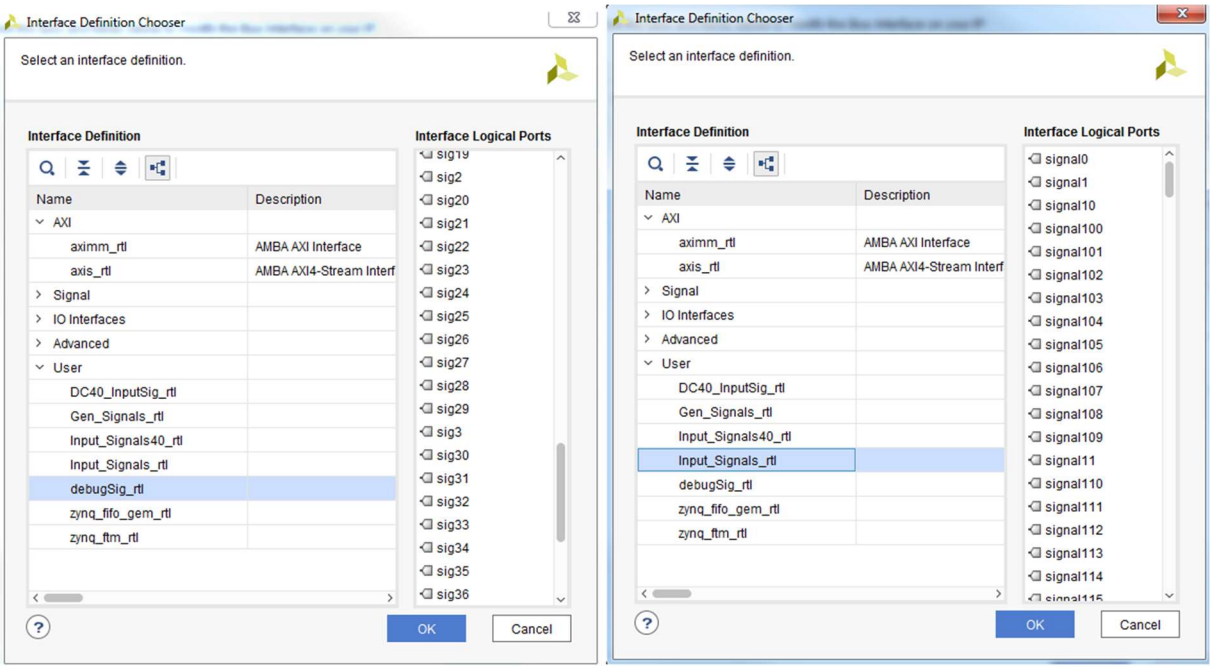


FIGURE 76: AID INTERFACES FOR 40 AND 300 INPUT SIGNALS

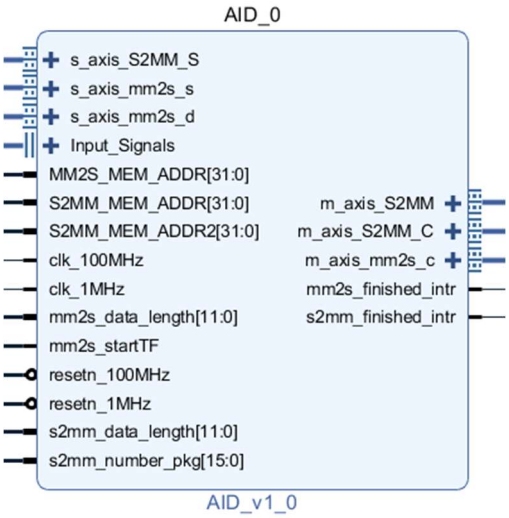


FIGURE 77: AID IP CORE WITH THE INPUTS AND OUTPUTS

The AID IP core uses both clocks, 1 MHz and 100 MHz, to work properly. The mm2s_startTF signal initiates the MM2S data transfer by sending the AXIS command data stream to the AXI DataMover. The command data is built with the start memory address and the data length. After sending the AXIS command data stream, the AXI DataMover reads the user data at the MM2S_MEM_ADDR from the RAM and sends it to the s_axis_mm2s_d interface of the AID module. The status of the transfer is sent to the s_axis_mm2s_s interface. If the MM2S transfer was successful, the mm2s_finished_intr occurs.

The user data contains the control information to start the debugging process with the chosen debugging settings. When the AID Module starts working, it sends the AXIS command data stream to the AXI DataMover with the m_axis_S2MM_C interface. Furthermore, the signal samples are packaged together to an AXIS data stream. This data stream is sent with the m_axis_S2MM interface to the AXI DataMover to write the data into the RAM. The AXI DataMover returns the transfer status to the s_axis_S2MM_S interface. If the S2MM transfer was successful, the internal counter increases, until the s2mm_number_pkg is reached. The s2mm_number_pkg is the number of signal samples, which are

collected for the UDP package and stored in the RAM, until the s2mm_finished_intr occurs. The Processing System handles the interrupts with the interrupt handlers.

To reduce the longest path, pipelines are used between the UNPKGModule and the DebugCoreModule. They are also used for the Input_Signals interface and between the DebugCoreModule and the PKG_Samples block. Pipelines are also used for the input signal interface. For the AID with 300 signals, the Pipeline300 is used and for the AID with 40 signals, the Pipeline40 is used. Figure 78 shows the structure of the whole AID IP core. An overview of the inputs and outputs of the AID IP core is shown in Table 51 and Table 52 in the Appendix.

The AID with 40 signals was tested on the ZedBoard and on the Controller Board with a Zynq 7000 xc7z100ffg900-2 FPGA. This FPGA is placed on a Trenz Board Zynq-7000 TE0782_100_2L SPRT PCB: REV02 on the Controller Board. The test results are discussed under the section testing the AID IP-Core in chapter 13 and chapter 14.

The AID with 300 signals was tested on the ZedBoard. The test results are discussed under the section testing the AID IP-Core in chapter 13 and chapter 14.

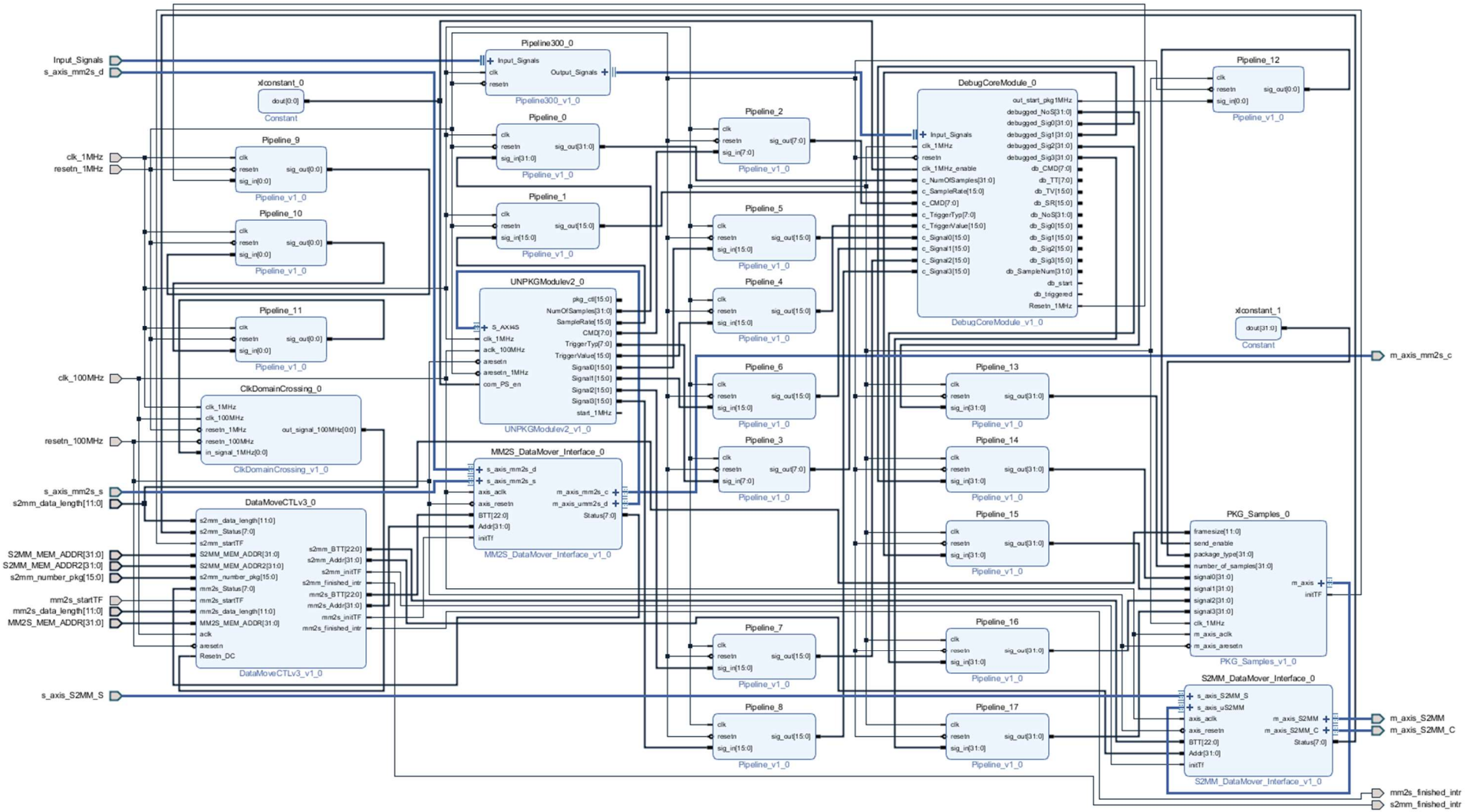


FIGURE 78: STRUCTURE OF THE PACKAGED AID IP CORE

5 COMPARISON OF THE AID40 AND AID300 RESOURCE UTILIZATION

In this section, the 2 AID IP cores are compared. The main focus is the FPGA resource utilization. Due to the 300 input signals, the AID300[36] needs a lot more resources on the FPGA than the AID40[37]. The 300 to 1 multiplexers of the signal selection blocks are the biggest parts. The smaller the number of the input signals, the lower is the resource usage on the FPGA. The first idea was, to debug 4 signals out 300 possible input signals, 32 bit each. Due to the already high FPGA utilization of the Controller Board with the inverter software, the input signals were reduced to 40.

Table 7 shows the FPGA utilization of the AID40 and AID300 on the ZedBoard. The AID40 needs 2055 of the available Slice LUTs, 1420 of the available Slice Registers, 755 of the available Slices, 2055 of the available LUT as Logic, 257 of the available LUT Flip Flop Pairs, 2 of the available Block RAM Tiles and 1 DSP, which is 0.455% of the available DSPs. The AID300 needs 5616 of the available Slice LUTs, 2500 of the available Slice Registers, 1730 of the available Slices, 5616 of the available LUT as Logic, 257 of the available LUT Flip Flop Pairs, 2 of the Block RAM Tiles and 1 of the available DSPs.

TABLE 3: COMPARISON OF THE FPGA RESOURCE UTILIZATION

Resource	Available on FPGA (ZedBoard)	AID40 used Resources	AID300 used Resources
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
Block RAM Tile	140	2	2
DSP	220	1	1

Table 8 shows the FPGA utilization of the AID40 and AID300 on the ZedBoard. In this table percentages are used. The values show how many of the total resources were used. The biggest differences of the AID40 and AID300 are at the Slice LUT, Slices and LUT as Logic. The AID300 uses 10,56% of the Slice LUT compared to the AID40 with 3,86%, 13,01% of the Slices compared to 5,68% and 10,56% of the LUT as Logic compared to 3,86%. These significant differences are mainly caused by the 300 to 1 multiplexer and the pipeline stage for the 300 input signal interface.

TABLE 4: COMPARISON OF THE FPGA RESOURCE UTILIZATION IN PERCENT

Resource	Resources on FPGA (ZedBoard)	AID40 used Resources	AID300 used Resources
Slice LUT	53200	3.86%	10.56%
Slice Register	106400	1.33%	2.35%
Slice	13300	5.68%	13.01%
LUT as Logic	53200	3.86%	10.56%
LUT Flip Flop Pairs	53200	0.48%	0.48%
Block RAM Tile	140	1.43%	1.43%
DSP	220	0.455%	0.455%

[36] AID300, Advanced Inverter Debugger IP core with 300 input signals
[37] AID40, Advanced Inverter Debugger IP core with 40 input signals

In both FPGA designs, the timing constraints [8] are fulfilled as shown in the Figures 79 and 80.

As long as the Worst Negative Slack (WNS) is positive, the path passes. If it is negative the path fails. The Total Negative Slack (TNS) is the sum of the negative slack in the design. If it is positive, then there is negative slack in the design. If it is 0, the timing is met. The TNS cannot be negative. The design passes the Worst Hold Slack (WHS), when it is positive. That means, there is no WHS in the design. If it is negative, the design fails. The Total Hold Slack (THS) is the sum of the WHS. If it is 0, the design passes, otherwise it fails. The Worst Pulse Width Slack (WPWS) checks if the periods of each clock pin is ok. If the value is positive, the design passes. If it is negative, the design fails. The Total Pulse Width Negative Slack (TPWS) is the sum of the WPWS. If it is 0, the design passes, otherwise it fails. The value cannot be negative.

In both Figures, the timing constrains are fulfilled. The WNS, WHS and WPWS are positive and the TNS, THS and TPWS are 0. This means the design passes the timing constrains.

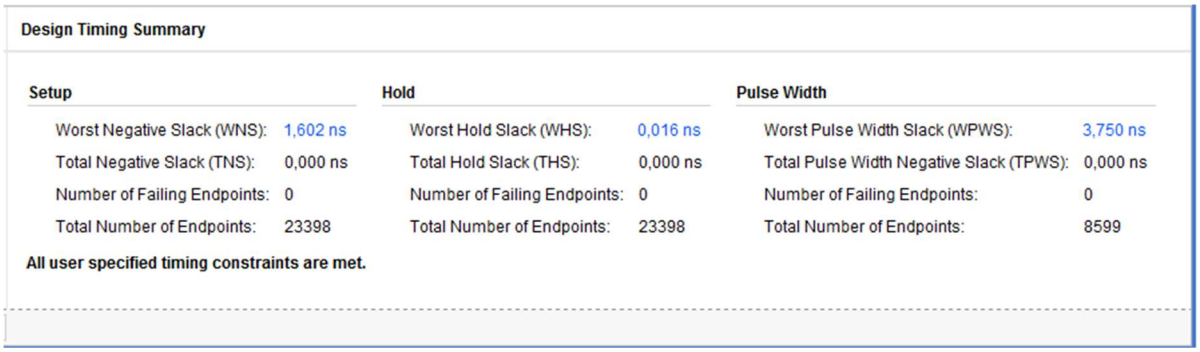


FIGURE 79: AID40 TIMING SUMMARY

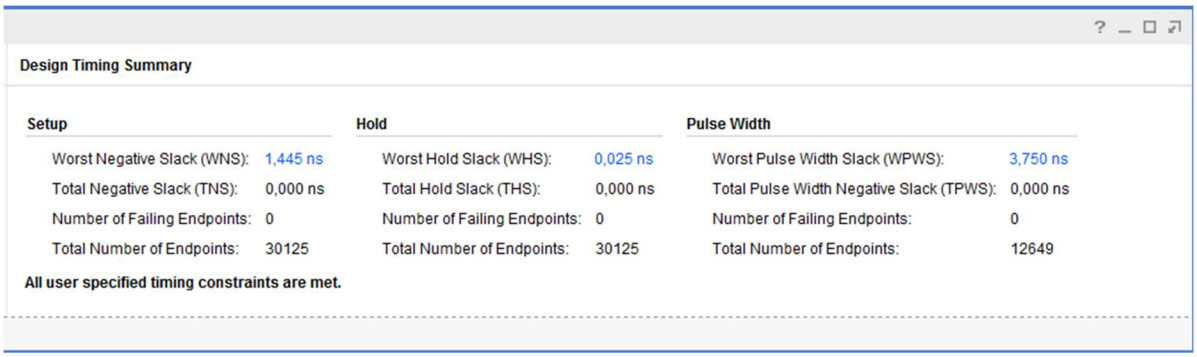


FIGURE 80: AID300 TIMING SUMMARY

Figure 81 shows the routing of the DebugCoreModule with 40 input signals. The white parts mark the used resources for the DebugCoreModule. The light blue areas mark the used resources of the whole AID with 40 input signals.

Figure 82 shows the routing of the DebugCoreModule with 300 input signals. The white parts mark the used resources for the DebugCoreModule. The light blue areas mark the used resources of the whole AID with 300 input signals. The areas for the AID300 and the DebugCoreModule are mainly bigger due to the 300 to 1 multiplexer and the Pipeline300 IP core.

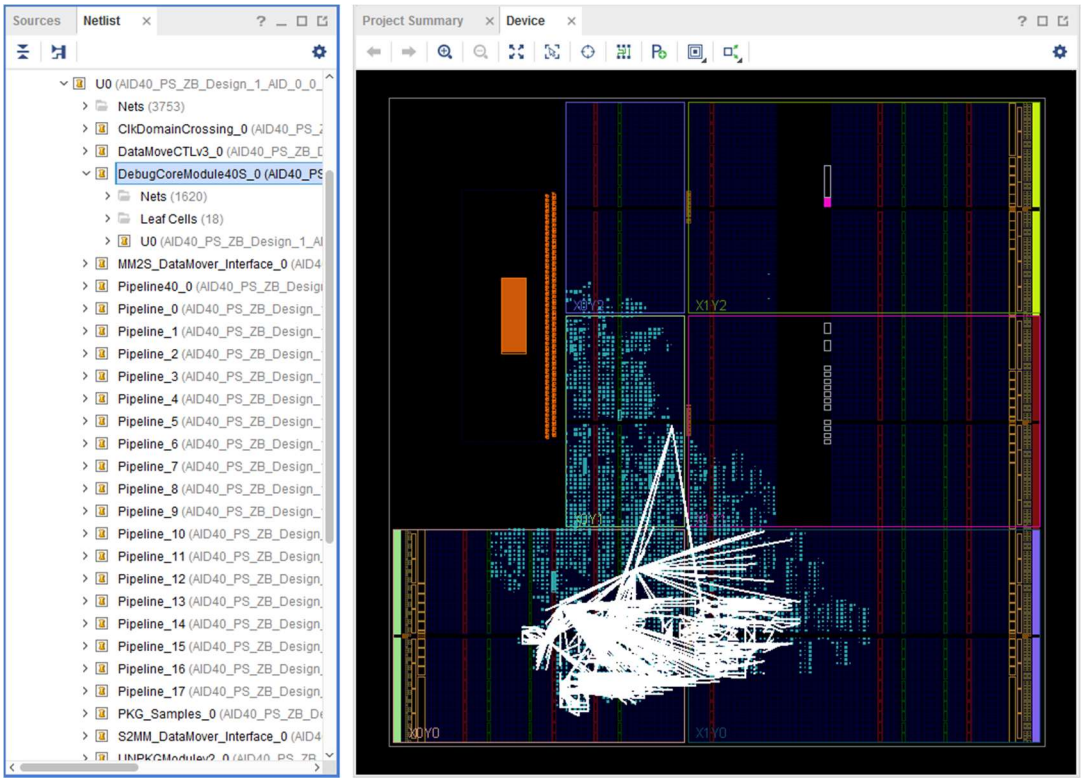


FIGURE 81: ROUTING OF THE DEBUGCOREMODULE WITH 40 SIGNALS ON THE ZEDBOARD

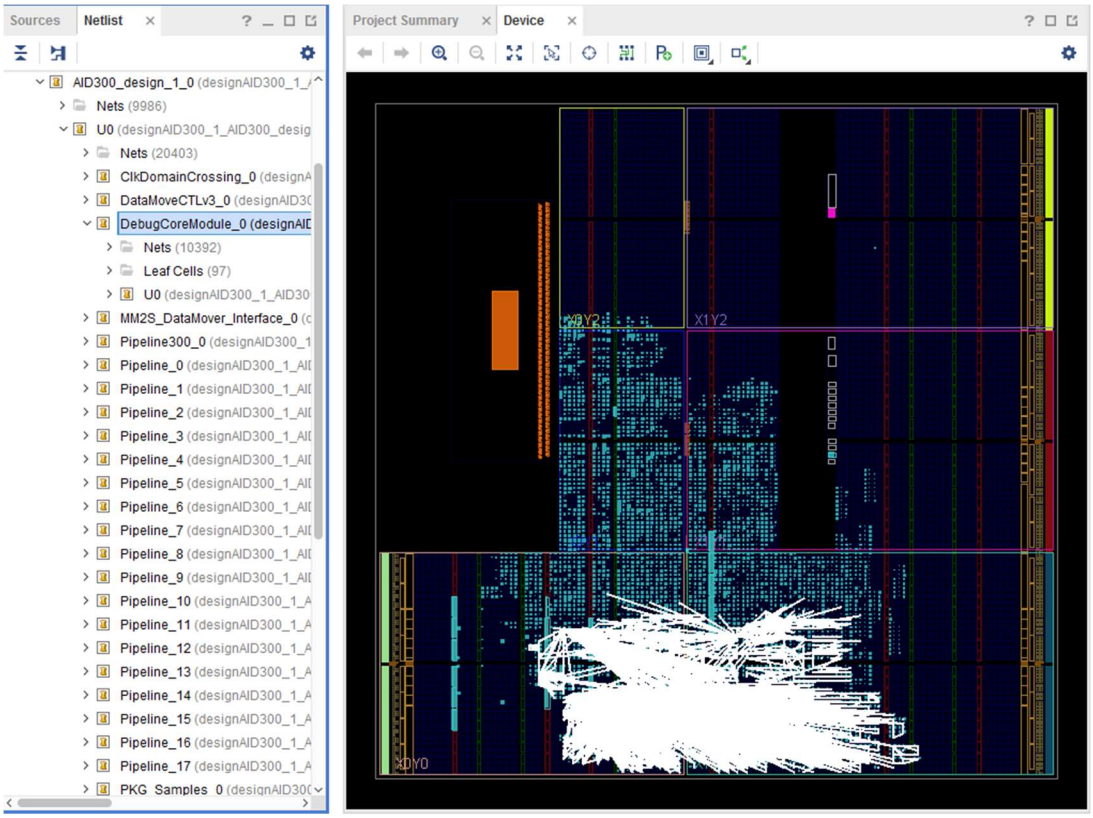


FIGURE 82: ROUTING OF THE DEBUGCOREMODULE WITH 300 SIGNALS ON THE ZEDBOARD

The Figures 83 and 84 show the power consumption of the FPGA designs with the AID40 and AID300. The power consumption is very similar. The PS needs in both designs 1.533W and is the biggest

consumer. All other parts are nearly the same. The BRAMs are a little bit different with 0.002W (AID40) and 0.013W (AID300).The clocks need a little bit more power, 0.032W with AID300 compared to 0.022W with AID40. The overall power consumption is with the AID300 higher than with AID40, 1.596W compared to 1.573W.

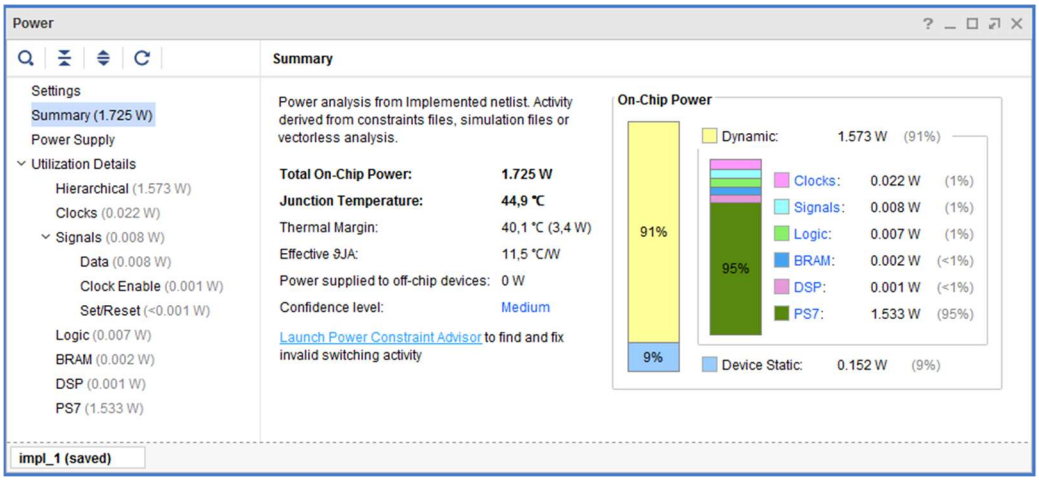


FIGURE 83: POWER CONSUMPTION OF THE AID40

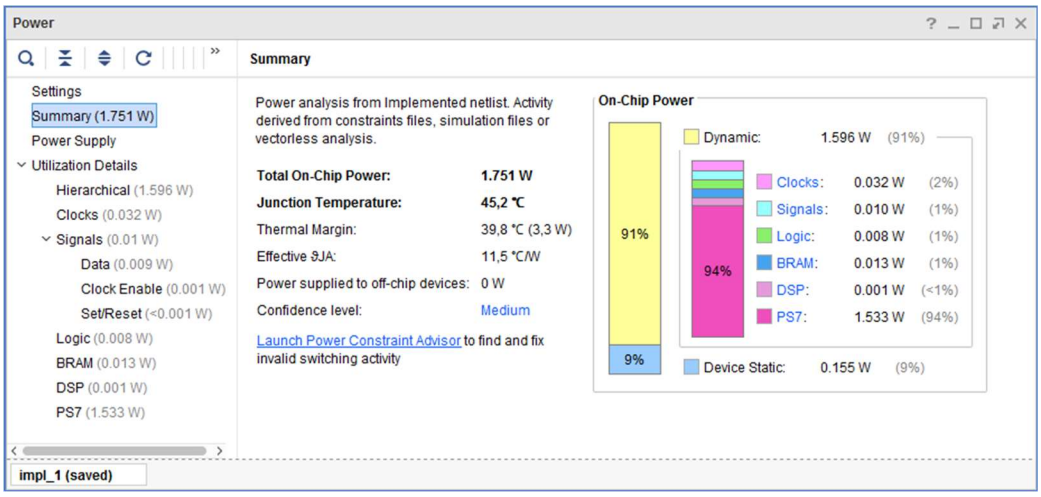


FIGURE 84: POWER CONSUMPTION OF THE AID300

6 DRIVER FILES FOR THE ZEDBOARD

After successfully running the synthesis, implementation and generation of the bit stream, the hardware with the bit stream was exported. Then, the Xilinx SDK (Xilinx Software Development Kit) can be used to write the driver files for the Processing System.

After starting the Xilinx SDK, the files for the hardware platform were automatically generated. A board support package was necessary. In the board support package settings, the lwip141 library was added to use the lwIP TCP/IP Stack (light weight TCP/IP Stack). The board support package settings are shown in Figure 85. The xilffs library was also added, but it is not used for the AID.

After creating the hardware platform files and the board support package an echo server application project was created. This project uses the hardware platform files. The echo server is used for the UDP/IP connection. The Program of the echo server was modified by adding an interrupt system, a logic control class and the class for the UDP connection.

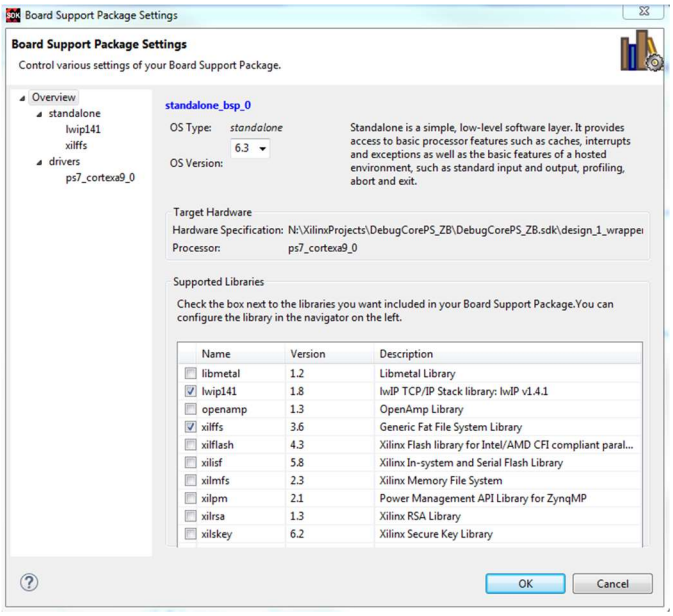


FIGURE 85: BOARD SUPPORT PACKAGE WITH LWIP141 LIBRARY

6.1 INTERRUPT_SYSTEM CLASS

The interrupt system class initializes the interrupt system and handles the different interrupts. It connects the interrupt controller to the interrupt handlers and maps the interrupts to the CPU. After that, the interrupts are enabled.

There are two different interrupts to handle, the MM2S_finished_intr and the S2MM_finished_intr. Each interrupt is handled with a different interrupt handler.

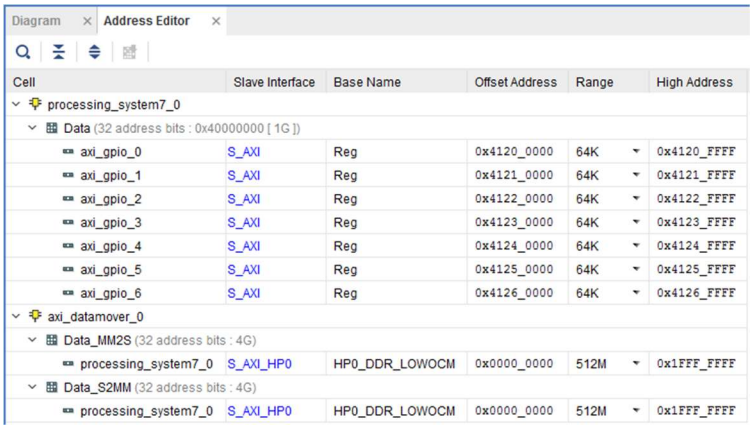
When an UDP package arrives, the Processing System checks if the package type is Start, Reset or GetVersion. If the packagetype is GetVersion, the PS sends the version number back to the user-interface. If the package type is Start or Reset, the control data is stored in the RAM and the AXI_GPIO [6] port for reading out the data of the RAM (mm2s_startTF) is set. This initializes the MM2S data transfer. After successfully reading out the data from the RAM with the AXI DataMover, the mm2s_finished_intr occurs. The interrupt controller handles the interrupt and the MM2S_Datamover_InterruptHandler is called. This handler resets the mm2s_startTF back to 0.

When the Debug-Core was started, the signal data are written into the RAM. When the number of samples for the UDP package is reached, the s2mm_finished_intr occurs. The interrupt controller calls the S2MM_Datamover_InterruptHandler, which reads the signal data out of the RAM, builds the UDP package and sends it to the user-interface.

6.2 LOGIC_CONTROL CLASS

The logic_control class is used to control the AXI_GPIO ports. The AXI_GPIO ports are directly connected to the Advanced Inverter Debugger IP core in the FPGA design. They are used to set the memory addresses, the data length of the S2MM/MM2S data transfer, the number of samples for the UDP package and to initialize the MM2S data transfer. The memory addresses of each AXI_GPIO port is used to set the different control signals.

Figure 86 shows the AXI_GPIO memory addresses in the Address Editor of Xilinx Vivado.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_1	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_2	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_gpio_3	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_4	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF
axi_gpio_5	S_AXI	Reg	0x4125_0000	64K	0x4125_FFFF
axi_gpio_6	S_AXI	Reg	0x4126_0000	64K	0x4126_FFFF
axi_datamover_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

FIGURE 86: AXI_GPIO MEMORY ADDRESSES

To access an AXI_GPIO port, a memory address is necessary. The memory addresses are mapped with the Address Editor to the different IP cores. With the Xilinx SDK, values can be assigned to the AXI_GPIO ports to use this values in the user logic. This is done with the following code:

```
Xil_Out32(GPIO_S2MM_NUMBER_PKG, sample_pkg_number);
```

Xil_Out32 is the function to access the memory with a 32-bit value. GPIO_S2MM_NUMBER_PKG is the memory address of the AXI_GPIO port, which is used to set the number of samples for the UDP package. The sample_pkg_number is the actual value. All of the AXI_GPIO ports are access in this way.

6.3 UPD_CON CLASS

The udp_con class handles the initialization of the UDP connection. It creates a new UDP connection and binds the listening port to the IP address.

The class also handles incoming UDP packages (control information for the AID). These data is received, ordered and written into the RAM. Functions from the LogicControl class are used to access the AXI_GPIO ports. After that, the MM2S data transfer is enabled. This class also handles the send process. Therefore, the UDP package is built with the signal data and sent to the user-interface. The following code shows the configuration of the AID, when a UDP package arrives:

```
/*
 * Set Properties for the MM2S transfer of the AXI DataMover
 * set mm2s memory address
 * set data length, which will be written from RAM
 */
SetMM2SAddress(MM2S_BASEADDR);
SetMM2SDataLength(EthBytesReceived);

// config S2MM Transfer
SetS2MMAddress(S2MM_BASEADDR);
SetS2MMAddress2(S2MM_BASEADDR2);
// set S2MM address to the first one
S2MMaddr1_active = 1;
// Length depends on package_type
SetS2MMDataLength(READ_LEN);
// Set number of samples for the UDP package
```

```
S2MMSetNumberOfPkgCollection(NUMBER_OF_PACKAGES);

// enable transfer (will cause the mm2s_intr, when the transfer was successful)
EnableMM2STransfer();
```

6.4 MAIN CLASS

The main function, starts the program. It was automatically created with the lwIP echo server application project. It initializes the lwIP stack with the MAC and IP addresses and adds the network interface to the network interface list. After that, the MM2S transfer is disabled and the interrupt system is initialized. The initialization of the interrupt system is shown in the following code:

```
// disable mm2s transfer, should be activated when udp data is coming
DisableMM2STransfer();

// init interrupt system
int Status = InitInterruptSystem(DEVICE_ID);
if (Status != XST_SUCCESS){
        xil_printf("error: init interrupt system failed.\n\r");
        return XST_FAILURE;
}
else
        xil_printf("init interrupt system done.\n\r");
```

After successfully initializing the interrupt system, a new DHCP server is created for the network interface by using the lwIP stack. The IP and MAC addresses are mapped to the server.

Then, the memory addresses are assigned. One is used for the MM2S transfer and two are used for the S2MM transfers with the switched buffer. After that, the UDP connection is initialized. This is shown in the following code:

```
// define memory location for buffer
DMA_MM2S_Buffer = (u8*) MM2S_BASEADDR;
S2MMaddr1_active = 1;
DMA_S2MM_Buffer32 = (u32*) S2MM_BASEADDR;
DMA_S2MM_Buffer232 = (u32*) S2MM_BASEADDR2;

// Start udp
InitUDP();

// receive and process packets
while (1) {

        xemacif_input(echo_netif);
}
```

After initializing the UDP connection with the adjusted ports, the DHCP server works as an echo server. The echo server constantly runs in a while loop and when a UDP package arrives the udp_recv_data function is called to process the incoming data.

The main.h file is used, to include all the different header files for the program. It also contains the constant definitions, like memory addresses, port numbers, interrupt ids, and other important variables for the correct work of the program. The different parameters, which are used for the constant definitions are mostly used from the xparameters.h file. The Xilinx SDK automatically generates the xparameters.h file from the exported FPGA design. The following code shows the definitions.

```
// Device ID and interrupt
#define DEVICE_ID                   XPAR_SCUGIC_0_DEVICE_ID
#define INTC                        XScuGic
#define INTC_HANDLER                XScuGic_InterruptHandler
#define S2MM_INTR     XPAR_FABRIC_DATAMOVECTLV3_0_S2MM_FINISHED_INTR_INTR
#define MM2S_INTR     XPAR_FABRIC_DATAMOVECTLV3_0_MM2S_FINISHED_INTR_INTR
// RAM memory addresses
#define MM2S_BASEADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00020000)
#define MM2S_HIGHADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0003FFFF)
#define S2MM_BASEADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00040000)
#define S2MM_HIGHADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0005FFFF)
#define S2MM_BASEADDR2          (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00060000)
#define S2MM_HIGHADDR2          (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0007FFFF)
#define UDP_DATA_BASEADDR       (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00080000)
#define UDP_DATA_HIGHADDR       (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0009FFFF)
// GPIO memory addresses
#define GPIO_MM2S_ADDR              XPAR_AXI_GPIO_0_BASEADDR
#define GPIO_S2MM_ADDR              XPAR_AXI_GPIO_1_BASEADDR
#define GPIO_MM2S_EN_ADDR           XPAR_AXI_GPIO_2_BASEADDR
#define GPIO_MM2S_LEN_ADDR          XPAR_AXI_GPIO_3_BASEADDR
#define GPIO_S2MM_LEN_ADDR          XPAR_AXI_GPIO_4_BASEADDR
#define GPIO_S2MM_NUMBER_PKG        XPAR_AXI_GPIO_5_BASEADDR
#define GPIO_S2MM_ADDR2             XPAR_AXI_GPIO_6_BASEADDR
// Define ports and IP
#define TR_UDP_PORT 63999
#define RV_UDP_PORT 64000
#define MACADDR { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 }
```

7 DRIVER FILES FOR THE CONTROLLER BOARD

The Controller Board is the target hardware from AVL List GmbH with an ARM Zynq-7000 Kintex-7 FPGA (xc7z100ffg900). The AID will be used to debug inverters and controllers, which will run on this FPGA.

To use the AID on the Controller Board, the AID, the AXI_GPIOs and the AXI DataMover IP cores were added to the existing Controller Board FPGA design. The added IP cores are shown in Figure 87.

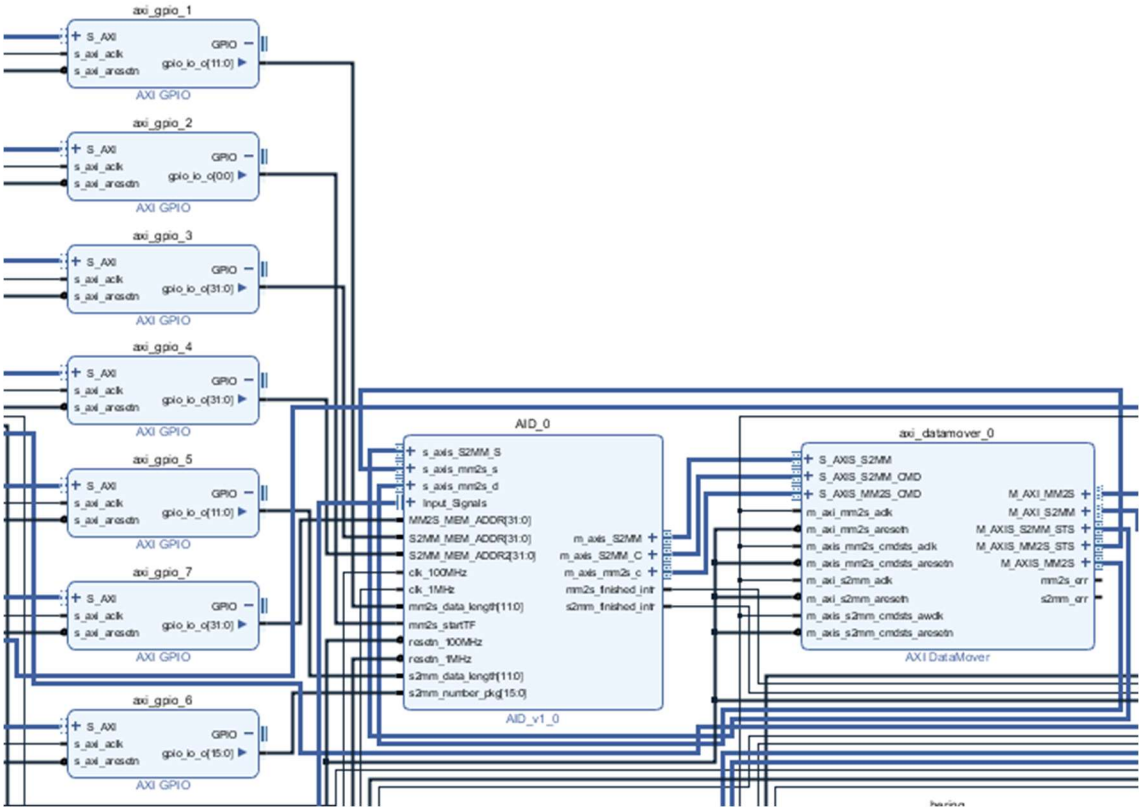


FIGURE 87: CONTROLLER BOARD WITH THE AID IP CORE

The AXI_GPIO memory addresses are different to the memory addresses of the ZedBoard design. Therefore, the driver files in the Xilinx SDK were modified and updated to the new values. The Address Editor is shown in Figure 88.

Diagram | Address Editor | IP Catalog

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G] ,0x80000000 [1G])					
bering/SAXI2SBUS_0	S_AXI	reg0	0x43C0_0000	64K	0x43C0_FFFF
com/SAXI2SBUS_0	S_AXI	reg0	0x43C1_0000	64K	0x43C1_FFFF
qm/SAXI2SBUS_0	S_AXI	reg0	0x43C3_0000	64K	0x43C3_FFFF
axi_gpio_1	S_AXI	Reg	0x4120_0000	32K	0x4120_7FFF
axi_gpio_2	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_3	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_gpio_4	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_5	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF
axi_gpio_6	S_AXI	Reg	0x4125_0000	64K	0x4125_FFFF
axi_gpio_7	S_AXI	Reg	0x4126_0000	64K	0x4126_FFFF
ctrl_board_interfaces_0	s00_axi	reg0	0x43C2_0000	64K	0x43C2_FFFF
isabella_genhdl_wrapper_DC_0	s00_axi	reg0	0x43C4_0000	256K	0x43C7_FFFF
axi_datamover_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
bering/axi_datamover_0					
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF

FIGURE 88: ADDRESS EDITOR OF THE CONTROLLER BOARD

After successfully running the synthesis, implementation and generation of the bit stream, the hardware with the bit stream was exported. To start the Xilinx SDK the TCL script build_swProj.tcl was used. This TCL script automatically generates the software project with all the included files in the Xilinx SDK from the Controller Board FPGA design.

In the application project tcp_ecat_server, a new folder with the name AID was added. In this AID folder all the necessary files were added to include the AID into the project.

The added files are:

- aid_intr_system.c
- aid_intr_system.h
- aid_settings.h
- logic_control.c
- logic_control.h
- udp_con.c
- udp_con.h

The board support package was also generated with the TCL script automatically. In case, the board support package is deleted, a new one with the included libraries lwip141 and xilffs has to be created. The lwip141 library is used for the lwIP TCP/IP Stack (light weight TCP/IP Stack) and the xilffs library is the Generic FAT File System library. Both libraries are used in the Controller Board project.

To include the AID files (from the ZedBoard) into the project with the Controller Board, little modifications were necessary. The modifications were, changing the name of the interrupt_system class to aid_intr_system class to avoid confusion with already existing similar classes and to change the main.h to aid_settings.h.

In the class aid_intr_system, a new function AIDConnect_InterruptHandler was added, which connects the AID interrupt handlers to the existing interrupt system of the design.

In the class udp_con the function InitUDP was renamed to InitAID, which initializes the UDP connection for the AID.

7.1 AID_INTR_SYSTEM CLASS

The class for the AID interrupt system handles the initialization of the interrupt system. It connects the interrupt controller to the interrupt handlers and maps the interrupts to the CPU. After that, the interrupts are enabled.

There are two different interrupts to handle, the MM2S_finished_intr and the S2MM_finished_intr. Each interrupt is handled with different interrupt handlers.

When an UDP package arrives, the Processing System checks if the package type is Start, Reset or GetVersion. If the package type is GetVersion, the PS sends the version number back to the user-interface. If the package type is Start or Reset, the control data are written into the RAM and the AXI_GPIO port for reading out the data of the RAM (mm2s_startTF) is set. This initializes the MM2S data transfer. After successfully reading out the data from the RAM with the AXI DataMover, the mm2s_finished_intr occurs. The interrupt controller handles the interrupt and the MM2S_Datamover_InterruptHandler is called, which resets the mm2s_startTF signal back to 0.

When the Debug-Core was started, the signal data is written into the RAM. After the number of samples for the UDP package is reached, the s2mm_finished_intr occurs. The interrupt controller calls the S2MM_Datamover_InterruptHandler, which reads the stored signal data out of the RAM, builds the UDP package and sends it to the user-interface.

Already existing project files initializes the interrupt system. Therefore, the InitInterruptSystem function from the AID_IntR_System class is not used. To connect the interrupt ids of the AID with the interrupt

system of the project, a new function was added. The AIDConnect_InterruptHandler function connects the existing interrupt controller with the interrupt ids and the interrupt handlers of the AID. Then, the AID is initialized with the function InitAID.

7.2 LOGIC_CONTROL CLASS

The logic_control class is used to control the AXI_GPIO ports. The AXI_GPIO ports are directly connected to the Advanced Inverter Debugger IP core in the FPGA design. They are used to set the memory addresses, the data length of the S2MM/MM2S data transfer, the number of samples for the UDP package and to initialize the MM2S data transfer. The memory addresses of each AXI_GPIO port is used to set the different control signals.

To access an AXI_GPIO port, a memory address is necessary. The memory addresses are mapped with the Address Editor to the different IP cores. With the Xilinx SDK, values can be assigned to the AXI_GPIO ports to use this values in the user logic. This is done with the following code:

```
Xil_Out32(GPIO_S2MM_NUMBER_PKG, sample_pkg_number);
```

Xil_Out32 is the function to access the memory with a 32-bit value. GPIO_S2MM_NUMBER_PKG is the memory address of the AXI_GPIO port, which is used to set the number of samples for the UDP package. The sample_pkg_number is the actual value. All of the AXI_GPIO ports are access in this way.

7.3 UPD_CON CLASS

The udp_con class handles the initialization of the UDP connection. It creates a new UDP connection and binds the ports to the IP address.

The class also handles incoming UDP packages (control information for the AID). These data is ordered and written into the RAM. Functions from the logic_control class are used to access the AXI_GPIO ports to set up the AID IP core. After that, the MM2S data transfer is enabled to start the data transfer from the RAM to the user logic. Depending on the command information the debugging process is started or stopped. The following code shows, how the AID IP core is set up.

```
/*
 * Set Properties for the MM2S transfer of the AXI DataMover
 * set mm2s memory address
 * set data length, which will be written from RAM
 */
SetMM2SAddress(MM2S_BASEADDR);
SetMM2SDataLength(EthBytesReceived);

// config S2MM Transfer
SetS2MMAddress(S2MM_BASEADDR);
SetS2MMAddress2(S2MM_BASEADDR2);
// set S2MM address to the first one
S2MMaddr1_active = 1;
// Length depends on package_type
SetS2MMDataLength(READ_LEN);
// Set number of packages for the UDP package
S2MMSetNumberOfPkgCollection(NUMBER_OF_PACKAGES);

// enable transfer (will cause the mm2s_intr, when the transfer was successful)
EnableMM2STransfer();
```

With the InitAID function, the UDP connection is initialized for the AID. The InitAID function creates a new UDP connection and bind the ports to the IP address. It also starts the UDP connection with the function StartUDP. With the DC_EN variable, the AID can be disabled. The InitAID function is called in the main.h. The following code shows, how the AID is initialized.

```
#if DC_EN
	// init AID
	print("Init AID UDP connection\n\r");
	AIDConnect_InterruptHandler(&InterruptController, CPU_ID);
	InitAID();

#endif
```

To receive UDP packages, a while loop in the main function checks, if UDP packages arrive. The incoming control information is used to set up the debugging process. The following code shows, how the UDP packages are received.

```
// in while loop
#if DC_EN
// enable AID debugger
	// receive udp data
	xemacif_input(echo_netif);

#endif
```

To send UDP packages, the SendDebuggedData32 function is used. This function is called in the S2MM_Datamover_InterruptHandler. The signal data is read from the RAM, the UDP payload is built and the UDP package is sent to the user-interface.

7.4 MAIN CLASS

The main function, starts the program. It initializes the lwIP stack with the MAC and IP addresses and adds the network interface to the network interface list. It also instantiates the interrupt system and enables the interrupts.

A new DHCP server is created for the network interface by using the lwIP stack. The IP and MAC addresses are mapped to the server.

With the InitAID function, the UDP connection of the AID is initialized. The InitAID function creates a new UDP connection and binds the ports to the IP address. With the DC_EN variable, the AID can be disabled. The following code shows, how the interrupts of the AID are connected to the interrupt system and how the AID is initialized.

```
#if DC_EN
	// init AID and connect interrupts
	AIDConnect_InterruptHandler(&InterruptController, CPU_ID);
	InitAID();

#endif
```

In the main function, EtherCat and other IP cores are set up for communication or I/O services. A while loop handles the communication with EtherCat, TCP and furthermore the new initialized UDP connection for the AID. To receive UDP packages, the while loop checks if UDP packages arrive. With incoming control information from the user-interface, the Processing System [1] starts or resets the Debug-Core.

The request for the version number is handled directly from the Processing System. The following code shows how UDP packages are received.

```
// in while loop
#if DC_EN
// enable AID debugger
        // receive udp data
        xemacif_input(echo_netif);

#endif
```

7.5 AID_SETTINGS

The aid_settings.h file is used, to include all the different header files for the AID into the application. It also contains the constant definitions like memory addresses, port numbers, interrupt ids, and other important variables for the correct work of the program. The different parameters, which are used as defines are mostly used from the xparameters.h file. The xparameters.h file is automatically generated by the Xilinx SDK. Some defines are shown below.

```
// Device ID and interrupt
#define DEVICE_ID               XPAR_SCUGIC_0_DEVICE_ID
#define CPU_ID                  XPAR_CPU_ID
#define S2MM_INTR               XPAR_FABRIC_AID_0_S2MM_FINISHED_INTR_INTR
#define MM2S_INTR               XPAR_FABRIC_AID_0_MM2S_FINISHED_INTR_INTR
// RAM memory addresses
#define MM2S_BASEADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00020000)
#define MM2S_HIGHADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0003FFFF)
#define S2MM_BASEADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00040000)
#define S2MM_HIGHADDR           (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0005FFFF)
#define S2MM_BASEADDR2          (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00060000)
#define S2MM_HIGHADDR2          (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0007FFFF)
#define UDP_DATA_BASEADDR       (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x00080000)
#define UDP_DATA_HIGHADDR       (XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0x0009FFFF)

// GPIO memory addresses
#define GPIO_MM2S_ADDR          XPAR_AXI_GPIO_7_BASEADDR
#define GPIO_S2MM_ADDR          XPAR_AXI_GPIO_3_BASEADDR
#define GPIO_MM2S_EN_ADDR       XPAR_AXI_GPIO_2_BASEADDR
#define GPIO_MM2S_LEN_ADDR      XPAR_AXI_GPIO_1_BASEADDR
#define GPIO_S2MM_LEN_ADDR      XPAR_AXI_GPIO_5_BASEADDR
#define GPIO_S2MM_NUMBER_PKG    XPAR_AXI_GPIO_6_BASEADDR
#define GPIO_S2MM_ADDR2         XPAR_AXI_GPIO_4_BASEADDR
// Define port to listen on
#define TR_UDP_PORT 63999
#define RV_UDP_PORT 64000
#define MACADDR { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 }
```

With help of the aid_settings.h, the UDP echo server can be modified. It is possible to disable the checksum. This increases the performance of the Processing System. The code is shown below.

```
/*
 * increase speed of udp
 * 1. remove section in code for tcp/ip
 * 2. manually assigning MAC and IP -> not 15-20 sec bootup for DHCP
 * 3. reduce overhead for checksum
 * 4. reduce overhead for checksum
 * 5. reduce overhead for checksum
 */
#undef LWIP_TCP
#undef LWIP_DHCP
//#undef CHECKSUM_CHECK_UDP
//#undef LWIP_CHECKSUM_ON_COPY
//#undef CHECKSUM_GEN_UDP
```

8 SIGNAL CONFIGURATION FILE GENERATOR

The signal configuration file generator generates the signal configuration file and was developed with C# and Microsoft Visual Studio. As mentioned in the section Packaged AID, the AID uses an interface to route the signals for the debugging process into the IP core. This interface name and the vhd file of the FPGA design with the included AID are used to generate the signal configuration file. This file maps the signal names to the input ports of the AID. The output file is a csv file, which is used for the user-interface of the AID. The signal configuration file generator is shown in Figure 89.

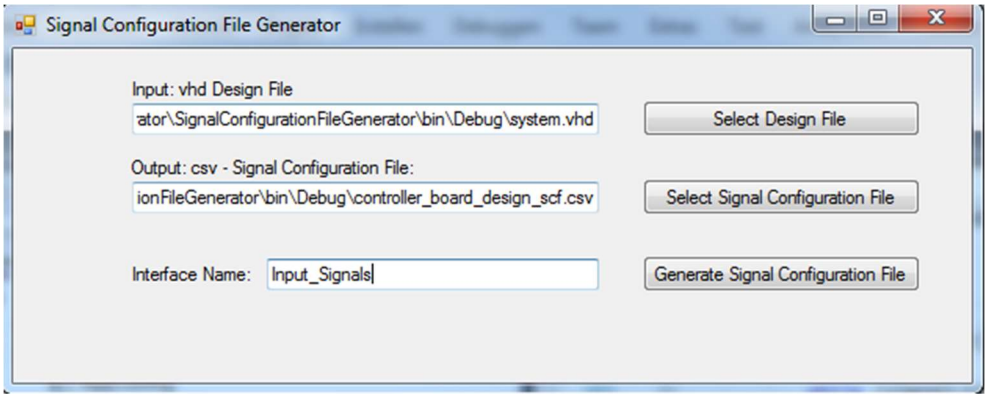


FIGURE 89: SIGNAL CONFIGURATION FILE GENERATOR

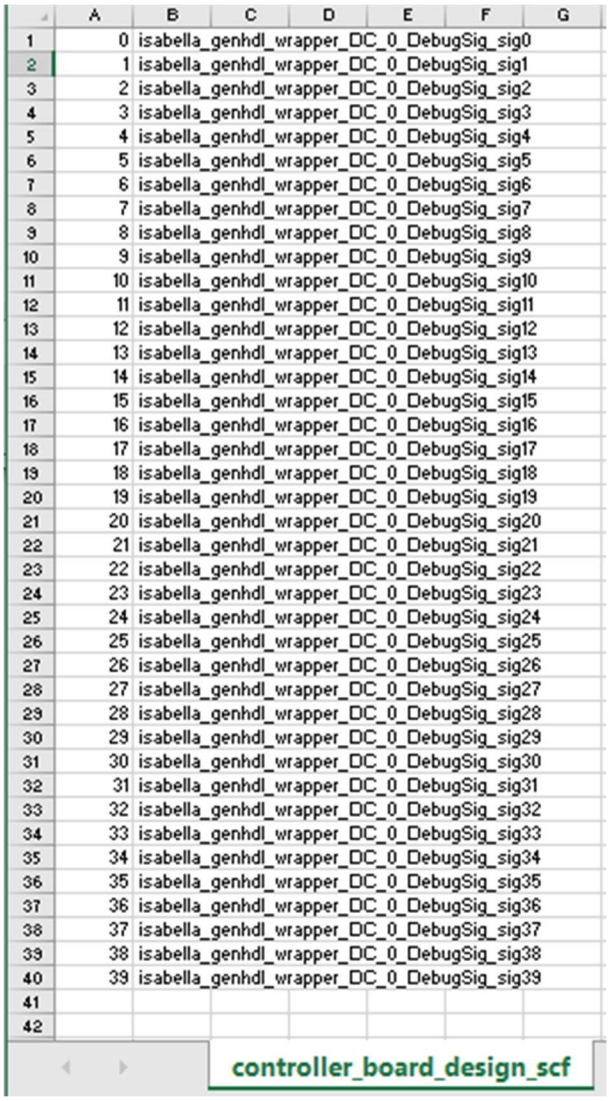
The vhd file of the FPGA design is selected with the Select Design File button. Then, the interface name of the AID is entered. The interface name of the AID40 and the AID300 is different. Therefore, it is important to enter the correct interface name, otherwise the mapping from the signal names to the input ports will not work. An output file is selected with the Select Signal Configuration File button. With the Generate Signal Configuration File button, the signal names are mapped to the AID input ports and saved in the output csv file.

Figure 90 shows a part of the vhd file with the FPGA design for the Controller Board. This part shows the port mapping of the AID IP core. The interface of the AID is called Input_Signals and the input ports are named sig0 to sig39 (AID with 40 signals). The signals, which should be debugged are from the interface Isabella_genhdl_wrapper_DC_0_DebugSig. These signals are from a black box, which contains custom inverters to test the AID. These inverters generate different sinus signals.

```
18519    sensor0_vib2_adc7923_sclk <= ctrl_board_interfaces_0_sensor_vib2_adc7923_sclk;
18520    sensor0_vib2_adc7923_ssel <= ctrl_board_interfaces_0_sensor_vib2_adc7923_ssel;
18521    sensor0_vib3_adc7923_mosi <= ctrl_board_interfaces_0_sensor_vib3_adc7923_mosi;
18522    sensor0_vib3_adc7923_sclk <= ctrl_board_interfaces_0_sensor_vib3_adc7923_sclk;
18523    sensor0_vib3_adc7923_ssel <= ctrl_board_interfaces_0_sensor_vib3_adc7923_ssel;
18524  AID_0: component system_AID_0_1
18525      port map (
18526       Input_Signals_sig0(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig0(31 downto 0),
18527       Input_Signals_sig1(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig1(31 downto 0),
18528       Input_Signals_sig10(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig10(31 downto 0),
18529       Input_Signals_sig11(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig11(31 downto 0),
18530       Input_Signals_sig12(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig12(31 downto 0),
18531       Input_Signals_sig13(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig13(31 downto 0),
18532       Input_Signals_sig14(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig14(31 downto 0),
18533       Input_Signals_sig15(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig15(31 downto 0),
18534       Input_Signals_sig16(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig16(31 downto 0),
18535       Input_Signals_sig17(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig17(31 downto 0),
18536       Input_Signals_sig18(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig18(31 downto 0),
18537       Input_Signals_sig19(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig19(31 downto 0),
18538       Input_Signals_sig2(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig2(31 downto 0),
18539       Input_Signals_sig20(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig20(31 downto 0),
18540       Input_Signals_sig21(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig21(31 downto 0),
18541       Input_Signals_sig22(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig22(31 downto 0),
18542       Input_Signals_sig23(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig23(31 downto 0),
18543       Input_Signals_sig24(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig24(31 downto 0),
18544       Input_Signals_sig25(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig25(31 downto 0),
18545       Input_Signals_sig26(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig26(31 downto 0),
18546       Input_Signals_sig27(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig27(31 downto 0),
18547       Input_Signals_sig28(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig28(31 downto 0),
18548       Input_Signals_sig29(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig29(31 downto 0),
18549       Input_Signals_sig3(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig3(31 downto 0),
18550       Input_Signals_sig30(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig30(31 downto 0),
18551       Input_Signals_sig31(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig31(31 downto 0),
18552       Input_Signals_sig32(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig32(31 downto 0),
18553       Input_Signals_sig33(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig33(31 downto 0),
18554       Input_Signals_sig34(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig34(31 downto 0),
18555       Input_Signals_sig35(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig35(31 downto 0),
18556       Input_Signals_sig36(31 downto 0) => isabella_genhdl_wrapper_DC_0_DebugSig_sig36(31 downto 0),
```

FIGURE 90: VHD FILE WITH THE FPGA DESIGN OF THE CONTROLLER BOARD

Figure 91 shows the generated signal configuration file with the input ports mapped to the signal names. This file is used to load the signal names into the AID user-interface. It is also possible to use this file generator for other IP cores, when the interface is known.



	A	B
1	0	isabella_genhdl_wrapper_DC_0_DebugSig_sig0
2	1	isabella_genhdl_wrapper_DC_0_DebugSig_sig1
3	2	isabella_genhdl_wrapper_DC_0_DebugSig_sig2
4	3	isabella_genhdl_wrapper_DC_0_DebugSig_sig3
5	4	isabella_genhdl_wrapper_DC_0_DebugSig_sig4
6	5	isabella_genhdl_wrapper_DC_0_DebugSig_sig5
7	6	isabella_genhdl_wrapper_DC_0_DebugSig_sig6
8	7	isabella_genhdl_wrapper_DC_0_DebugSig_sig7
9	8	isabella_genhdl_wrapper_DC_0_DebugSig_sig8
10	9	isabella_genhdl_wrapper_DC_0_DebugSig_sig9
11	10	isabella_genhdl_wrapper_DC_0_DebugSig_sig10
12	11	isabella_genhdl_wrapper_DC_0_DebugSig_sig11
13	12	isabella_genhdl_wrapper_DC_0_DebugSig_sig12
14	13	isabella_genhdl_wrapper_DC_0_DebugSig_sig13
15	14	isabella_genhdl_wrapper_DC_0_DebugSig_sig14
16	15	isabella_genhdl_wrapper_DC_0_DebugSig_sig15
17	16	isabella_genhdl_wrapper_DC_0_DebugSig_sig16
18	17	isabella_genhdl_wrapper_DC_0_DebugSig_sig17
19	18	isabella_genhdl_wrapper_DC_0_DebugSig_sig18
20	19	isabella_genhdl_wrapper_DC_0_DebugSig_sig19
21	20	isabella_genhdl_wrapper_DC_0_DebugSig_sig20
22	21	isabella_genhdl_wrapper_DC_0_DebugSig_sig21
23	22	isabella_genhdl_wrapper_DC_0_DebugSig_sig22
24	23	isabella_genhdl_wrapper_DC_0_DebugSig_sig23
25	24	isabella_genhdl_wrapper_DC_0_DebugSig_sig24
26	25	isabella_genhdl_wrapper_DC_0_DebugSig_sig25
27	26	isabella_genhdl_wrapper_DC_0_DebugSig_sig26
28	27	isabella_genhdl_wrapper_DC_0_DebugSig_sig27
29	28	isabella_genhdl_wrapper_DC_0_DebugSig_sig28
30	29	isabella_genhdl_wrapper_DC_0_DebugSig_sig29
31	30	isabella_genhdl_wrapper_DC_0_DebugSig_sig30
32	31	isabella_genhdl_wrapper_DC_0_DebugSig_sig31
33	32	isabella_genhdl_wrapper_DC_0_DebugSig_sig32
34	33	isabella_genhdl_wrapper_DC_0_DebugSig_sig33
35	34	isabella_genhdl_wrapper_DC_0_DebugSig_sig34
36	35	isabella_genhdl_wrapper_DC_0_DebugSig_sig35
37	36	isabella_genhdl_wrapper_DC_0_DebugSig_sig36
38	37	isabella_genhdl_wrapper_DC_0_DebugSig_sig37
39	38	isabella_genhdl_wrapper_DC_0_DebugSig_sig38
40	39	isabella_genhdl_wrapper_DC_0_DebugSig_sig39
41		
42		

controller_board_design_scf

FIGURE 91: GENERATED SIGNAL CONFIGURATION FILE FOR THE CONTROLLER BOARD

To create the Signal Configuration File Generator, different classes were used:

- The class Program.cs instantiates the Signal Configuration File Generator window and runs it.
- The class Form1.cs is the main window for the Signal Configuration File Generator. It contains the buttons and other elements. It instantiates the UIHandler.
- The class UIHandler.cs handles the inputs of the user-interface and checks them for validity. It also reads the vhd file and generates the csv file.
- The class SignalEntry is used for mapping the signal names to the input ports. Therefore, it contains only two member variables, the signal name and the index.

9 LABVIEW USER-INTERFACE

9.1 GUI

The LabVIEW user-interface controls the Advanced Inverter Debugger on the FPGA. It sends the control information to the FPGA and receives the signal data. The user-interface is shown in Figure 92.

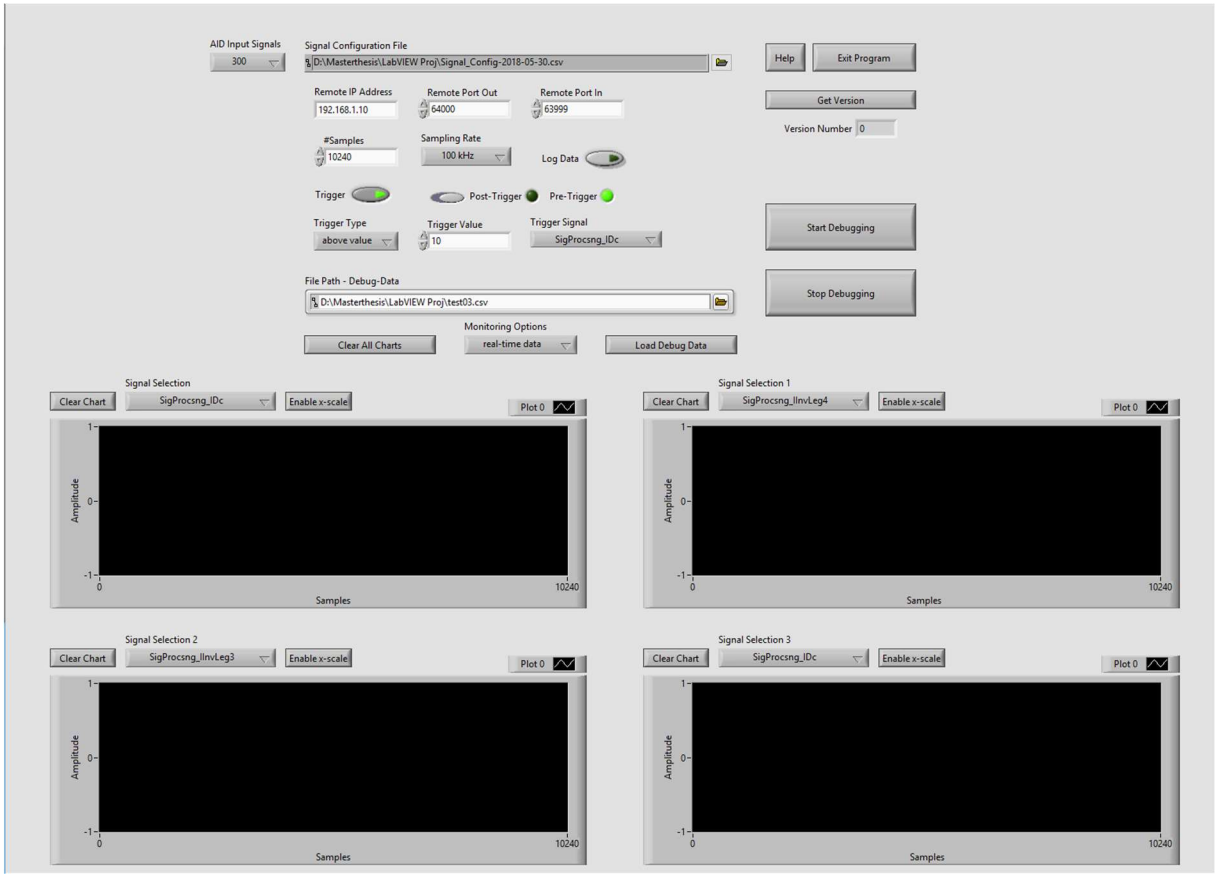


FIGURE 92: LABVIEW USER-INTERFACE TO CONTROL THE AID

The IP address and the port numbers are used to set up the UDP connection between the user-interface and the FPGA. The default IP address is 192.168.1.10, which was adjusted at the target HW. The default input port is 63999 and the default output port is 64000.

The signal configuration file is used to load the signal names of the FPGA design into the LabVIEW user-interface. The AID is available with 40 and 300 input signals. Therefore, the signal configuration files are different for both versions. After the signal configuration file is loaded, the signal selection menus are updated to the signal names. This provides a better overview of the connected signals and makes the interpretation of the debugged results easier.

The number of samples defines how long the AID samples, transmits and monitors the data. The number of samples is a multiple of 1024. The minimum number is 1024 and the maximum is 999424 samples. The Advanced Inverter Debugger automatically stops the debugging process, when the number of samples matches the adjusted number of samples from the user interface. When the AID is running in this mode, it is called normal mode. The second mode is the infinity mode. The AID is running as long as the stop debugging button is pressed. The stop debugging button transmits the Reset UDP package

(package type 3) to the FPGA. After receiving the stop information, the AID on the FPGA is reset. To enable the infinity mode, the number of samples has to be set to 0.

The sample frequency for the debugging process is determined by the sample rate menu. It was created with a Menu Ring. The advantage of this element is, that it is easy to handle and can be extended with more elements very easily.

The trigger is set up with the trigger settings. The Trigger button enables the trigger and by switching the shift button, the trigger can be switched between pre- and post-trigger. With the Trigger Type selection and the trigger value, the trigger event can be defined. The trigger signal determines the signal, which is used for the trigger. If the trigger is enabled, the selected signal is automatically updated in the signal selection menu of the first chart.

To log the received signal data, data logging can be enabled with the Log Data button. A csv file must be selected with the file path selection, File Path – Debug-Data. After starting the debugging process, the received signal data is written into the selected csv file and also monitored in the charts. This file path is also used to load the logged data. To load the saved data, the Monitoring Options has to be changed to logged data. After setting the right file path (csv file) and the monitoring options, the button Load Debug Data must be pressed. The button initializes the read process from the csv file into the LabVIEW application. When loading data from files into the LabVIEW application, it is important to set the corresponding signal configuration file. By setting the wrong signal configuration file, it can be, that the application can't find the signal name entries and the signal names are not updated correctly in the signal selection menus. However, the signal values are displayed in the charts.

To monitor the signal data, 4 charts are used. Each chart has its own signal selection menu to select the debugging signal for the chart. The first signal selection menu is disabled, when the trigger is active. In this case, the trigger signal menu is used as signal selection for chart 1.

To get a better overview of the monitored data, the x-axis can be changed. This is possible by pressing the button Enable x-scale. This enables the fields for setting the start and the end point on the x-axis. The Clear Chart button resets the history of the chart. There is also a Clear All Charts button, which resets the history of all charts.

To start the debugging process, the Start Debugging button has to be pressed. The user-interface sends the control information for starting the AID to the FPGA.

To stop the debugging process, the Stop Debugging button has to be pressed. The user-interface sends the control information for stopping/resetting the AID to the FPGA.

To get the version number, the Get Version button has to be pressed. The user-interface sends the control information (package type 5) to the FPGA, which returns the version number (package type 6). The received UDP package with the version number is not processed with LabVIEW. It appeared that the processing structure slowed down the receiving while loop and performance problems occurred. Therefore, this part was removed in the final version to gain more performance.

The close the LabVIEW-application, the Exit Program button has to be pressed.

If there is something not clear, how the AID is controlled by the LabVIEW user-interface, the Help button can be pressed. Under the different sections, the whole flow to set up a correct debugging process is explained step by step.

9.2 LABVIEW FUNCTIONS

There are several LabVIEW functions, which provide the functionality of the user-interface. Most functions run in separate while loops. To increase the performance, some while loops use a Wait(ms) function to reduce the number of executions. The free execution time is used for other while loops, like the receiving while loop, to process the incoming UDP data.

Figure 93 shows the while loop, to load the signal configuration data into the trigger and signal selection menus of the user-interface. First, the signal configuration file has to be selected. The while loop opens the file and reads line by line from the file. Each selection menu is filled with the read signal names. This while loop is processed every 50ms to increase the performance for other while loops.

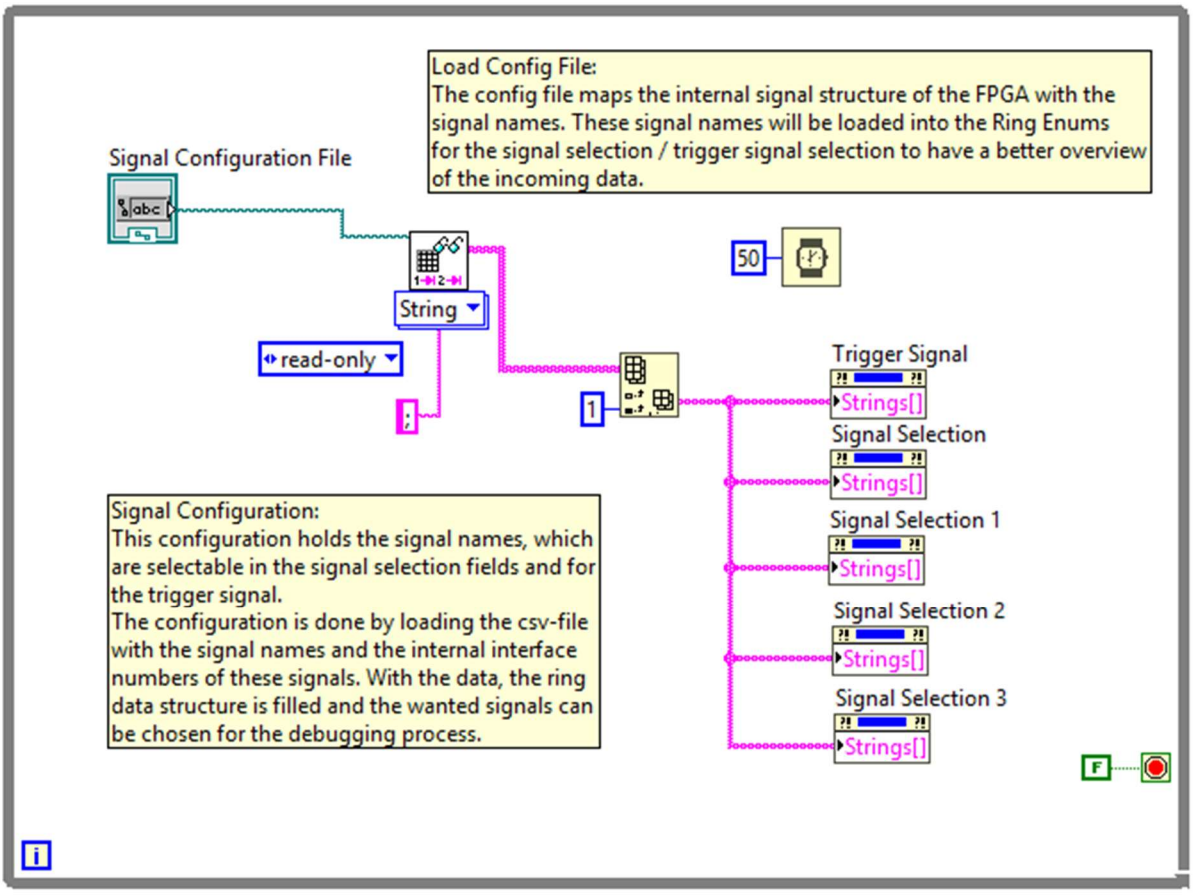


FIGURE 93: LOAD SIGNAL CONFIGURATION FILE

Figure 94 shows the while loop to process the x-axis scale, when it is enabled. Each chart has its own button to enable the x-axis scale.

When the x-axis scale is enabled, two new input fields appear, the x-min input field and the x-max input field. The x-min input field defines the minimum value of the x-axis and the x-max input field defines the maximum value of the x-axis. With this functionality, a scalability in the x-axis direction was implemented to get a better overview of each chart.

If the minimum value is greater or equal to the maximum value, the maximum value is automatically increased by 100.

When the x-axis scale is disabled, the input field for the minimum x-axis and the maximum x-axis are invisible. When the number of samples is 0, which means the AID is running in infinity mode, the maximum value of the x-axis is incremented (Figure 94, Chart1). If a finite number of samples is selected in the menu, the maximum value of the x-axis is set to the selected number of samples (Figure 94, Chart3).

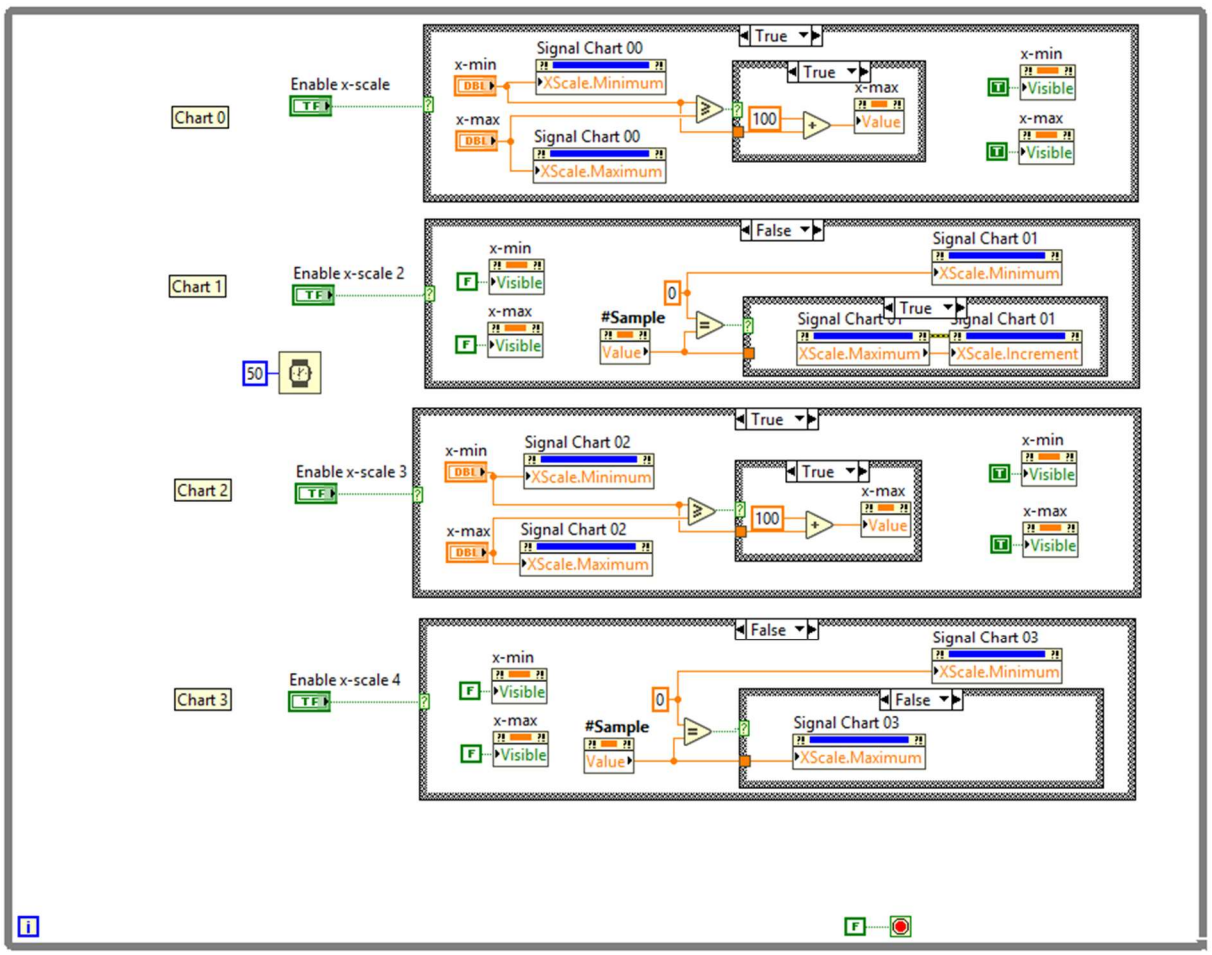


FIGURE 94: CHART X-AXIS SCALE

Figure 95 shows the while loop for resetting the charts. There is a Clear All Charts button, which resets the history of all signal charts. With the Clear Signal Chart buttons, every signal chart can be reset separately. The execution of this while loops is also slowed down by 50ms.

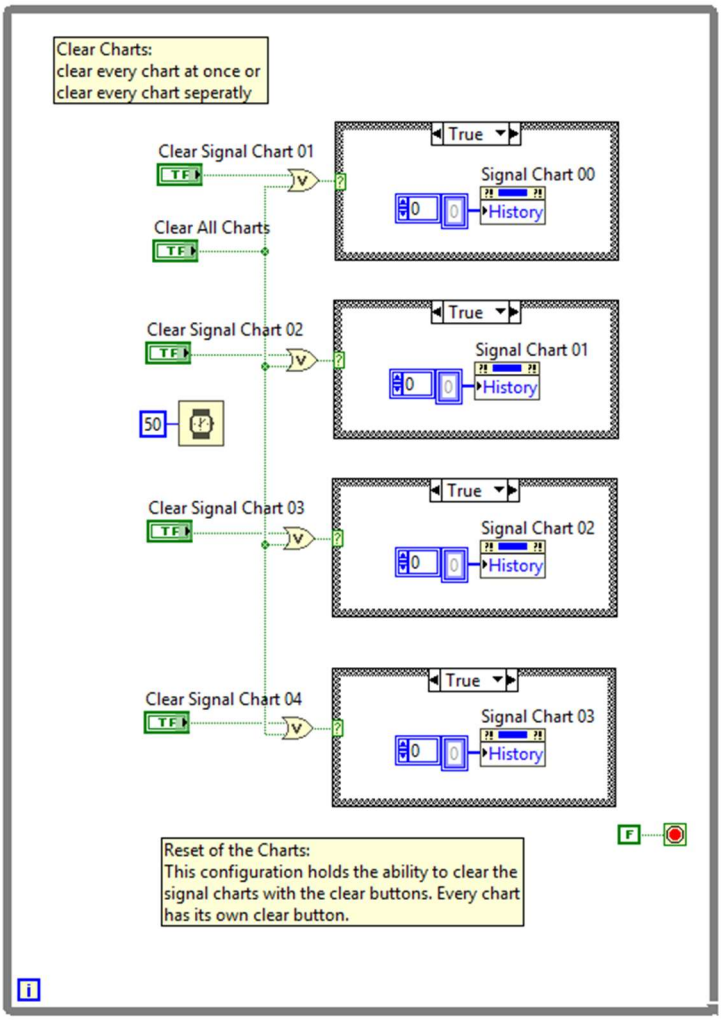


FIGURE 95: CLEAR SIGNAL CHARTS

Figure 96 shows the trigger event to close the user-interface. By pressing the Exit Program button, the trigger event occurs and the user-interface is closed.

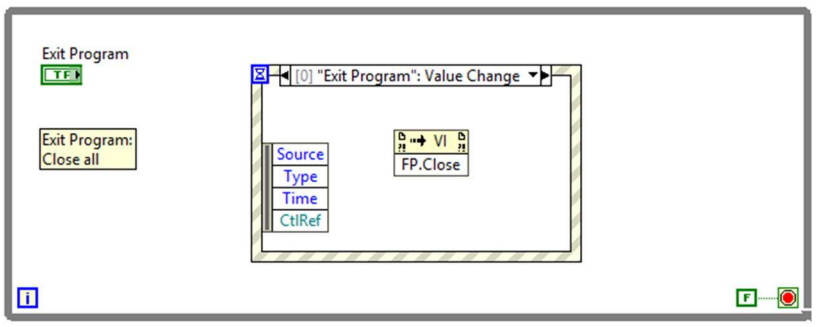


FIGURE 96: GUI EXIT

Figure 97 shows the while loop to load the debugged data from the log file. The log file is selected with the File Path – Debug-Data. After selecting the file, the monitoring options must be changed to logged data. To read the data from the file, the Load Debug Data button must be pressed. Then, the signal

charts are filled with the data and the trigger and signal selection menus are updated to the signal names, which were debugged in the log file. It is important, that the same signal configuration file is used, which was used to log the data, otherwise the signals can't be updated in the signal selection menus. However, the signal values are displayed.

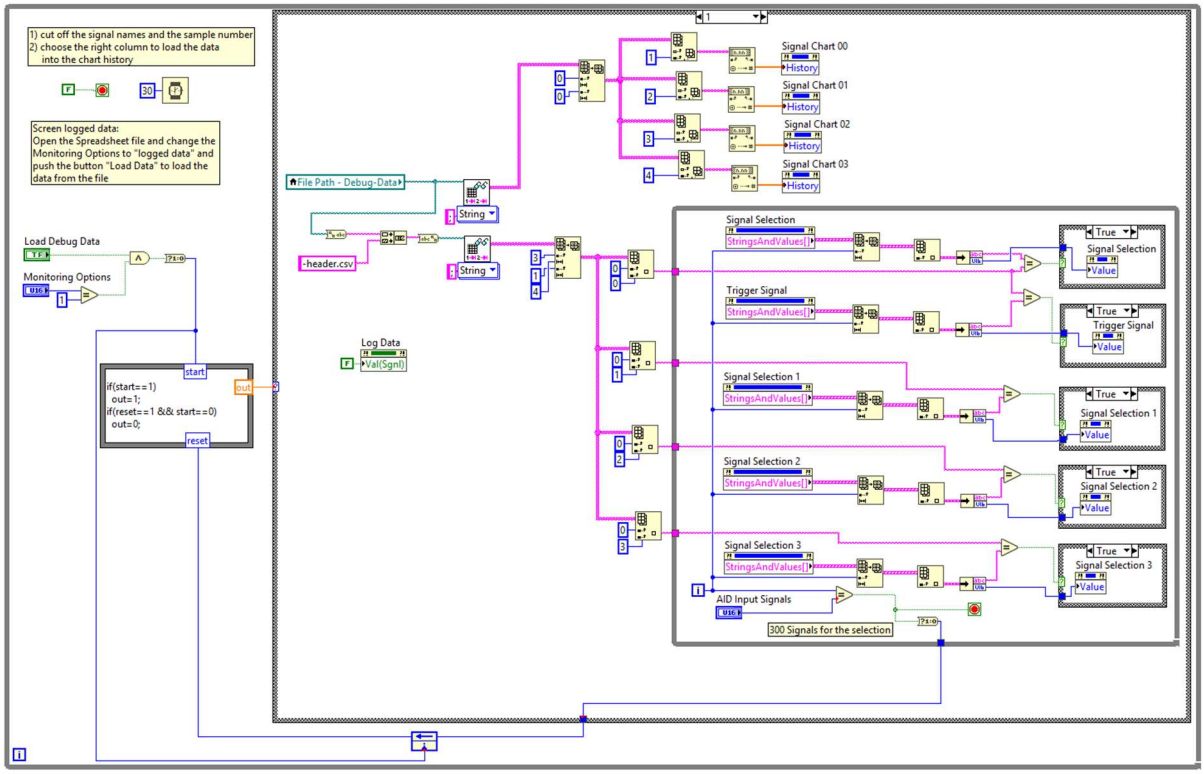


FIGURE 97: LOAD DEBUGGED DATA

Figure 98 shows the while loop for sending the command information to the FPGA. Before the command data is sent, the adjusted parameters are saved into the header csv file. There are 2 files which are generated with the LabVIEW user-interface when data logging is enabled. The normal data file and the header file. Due to complications with writing these data into one file, 2 separate files are created. In the header file, the debugging information is stored. The debugging information contains, time and date when the debugging started, sample rate, number of samples, trigger options, trigger value and the debugged signal names.

In the data file all of the received data is stored. This data contains the sample number and the values of the debugged signals. The execution of the while loop is also slowed down by 50ms.

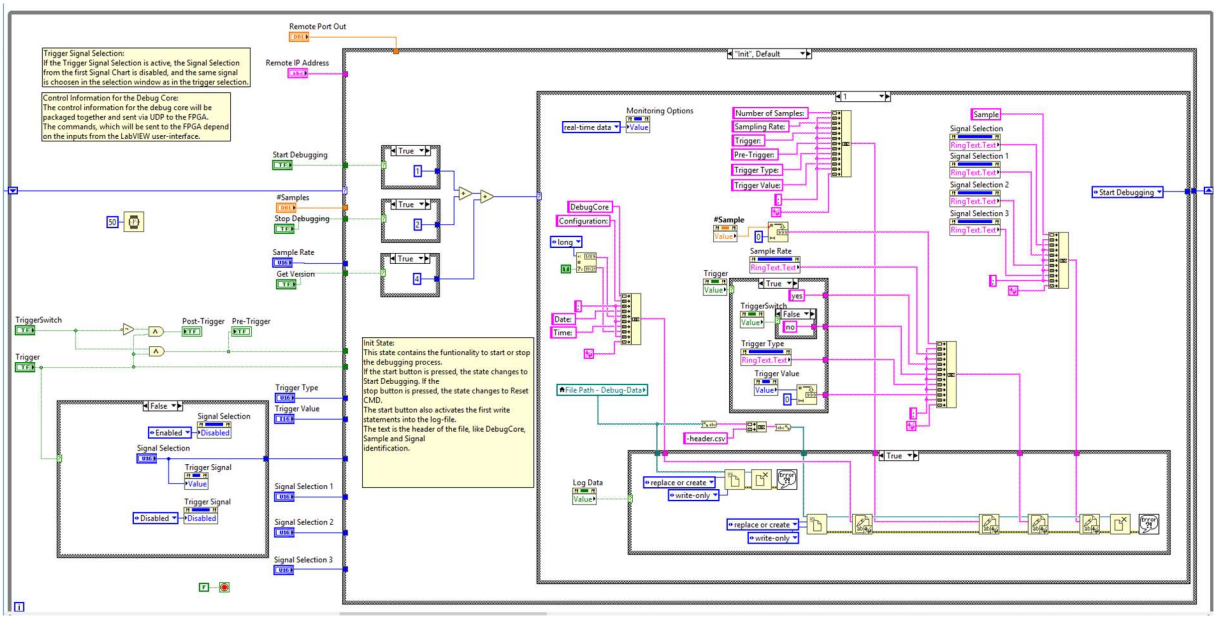


FIGURE 98: WRITE HEADER

Figure 99 shows the case, when the Start Debugging button is pressed. When data logging is active, the header file is created and after that, the state of the state machine changes to Start Debugging. When data logging is not active, the state machine goes right into the Start Debugging state, without creating the header file.

In this state, the UDP connection is established to send the control information to the Debug-Core. After sending the control information, the connection is closed again. The state machine switches into the Sending 0 and into the init state.

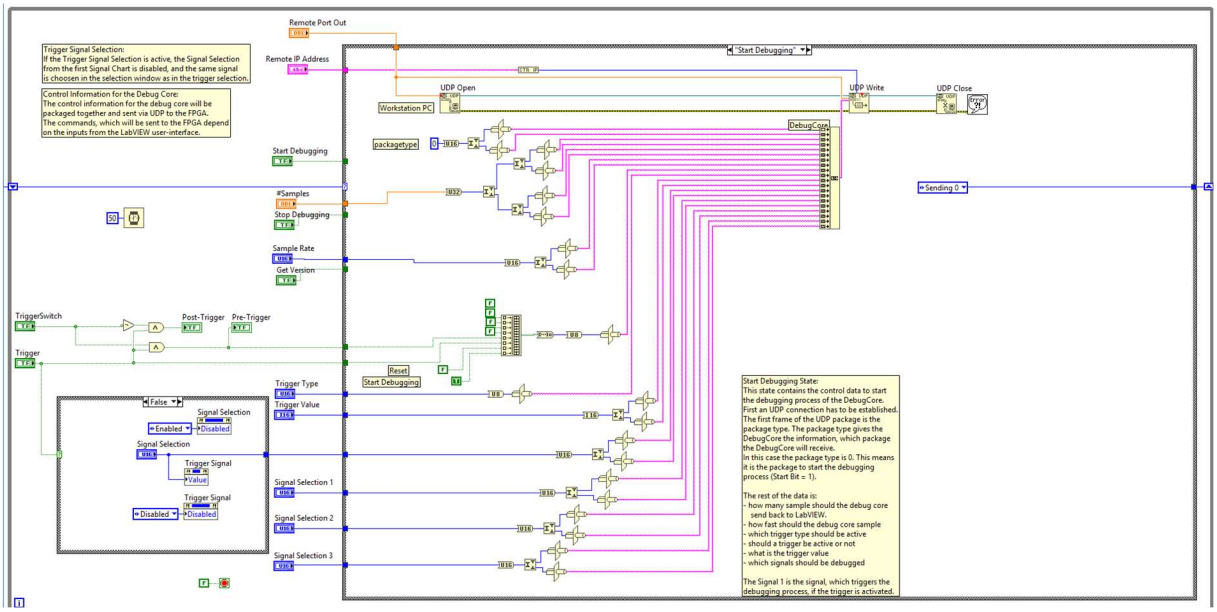


FIGURE 99: START DEBUGGING COMMAND

Figure 100 shows the Reset state. After pressing the Stop Debugging button, the state machine switches into the Reset state. The Reset state establishes an UDP connection to send the Reset command information to the Debug-Core. After sending the information, the state changes to Reset 0. In the Reset

0 state, the Reset command is sent again, to increase the possibility that the Debug-Core receives the reset command. Then, the state machine switches into the init state.

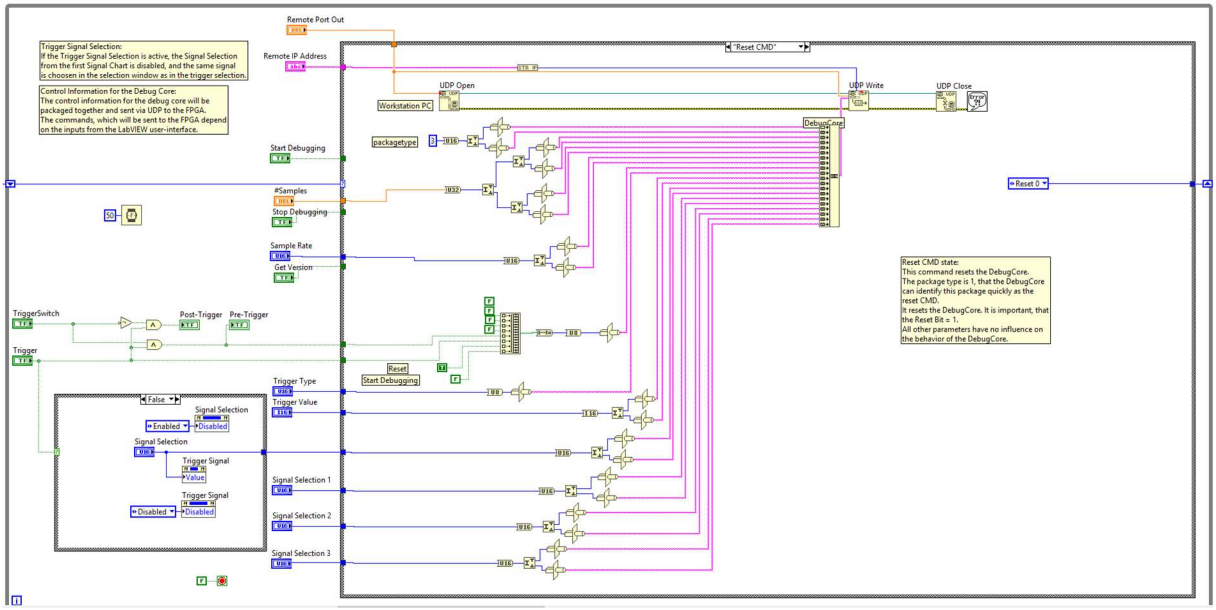


FIGURE 100: RESET COMMAND

Figure 101 shows the state Get Version. The state machine jumps into this state, when the Get Version button is pressed. An UDP connection is established and the GetVersion command information is sent to the Debug-Core. Due to performance issues, the receive part for the version number was removed. To receive the version number, the package type was compared to the version acknowledge package. If the package type was correct, the output field for the version number was updated with the received value. Due to this comparison, the performance dropped and this part was removed.

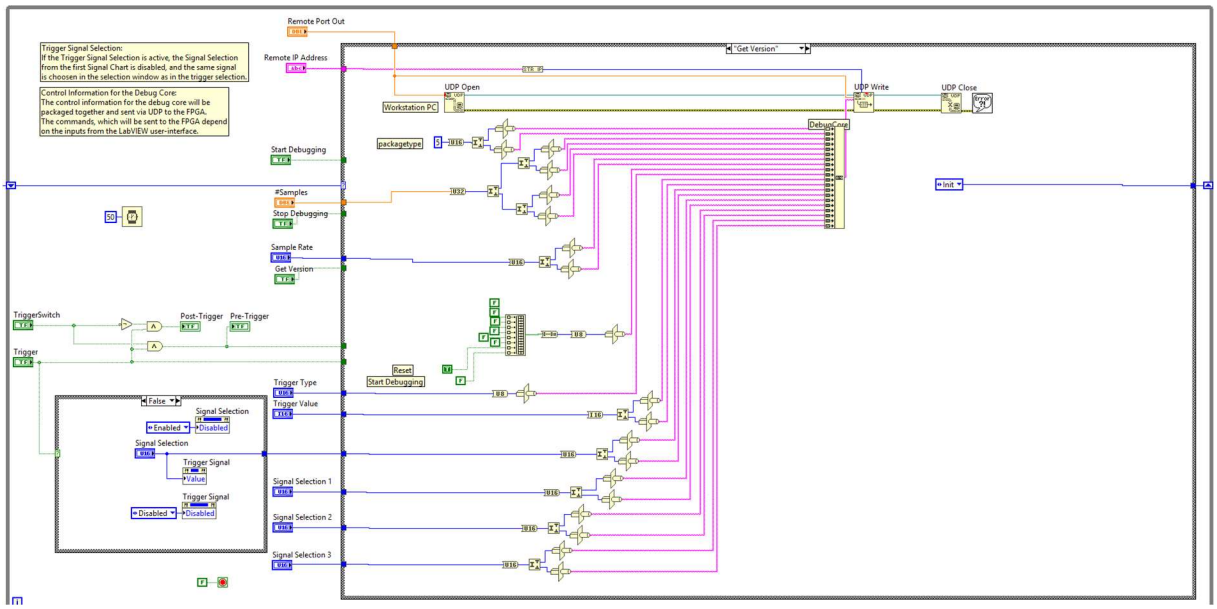


FIGURE 101: GET VERSION NUMBER

The Figures 102-104 show parts of the receiving process. The while loop contains elements for receiving and processing the UDP data.

First, the UDP connection on the receiver side is created. At that point, the file for the debugging process is opened to write the data into the file. Opening the file at this point saves time, because inside the while loop, the data only needs to be written into the file. Otherwise, the csv write function could be used, but that would slow down the performance. The received data is read from the UDP Receive block.

The data processing part splits the received data into the sample number and signal values. The signal values are added to arrays and displayed in the signal charts. Furthermore, the signal values and the number of samples are prepared for the csv file. When data logging is active, the prepared data is written into the log file.

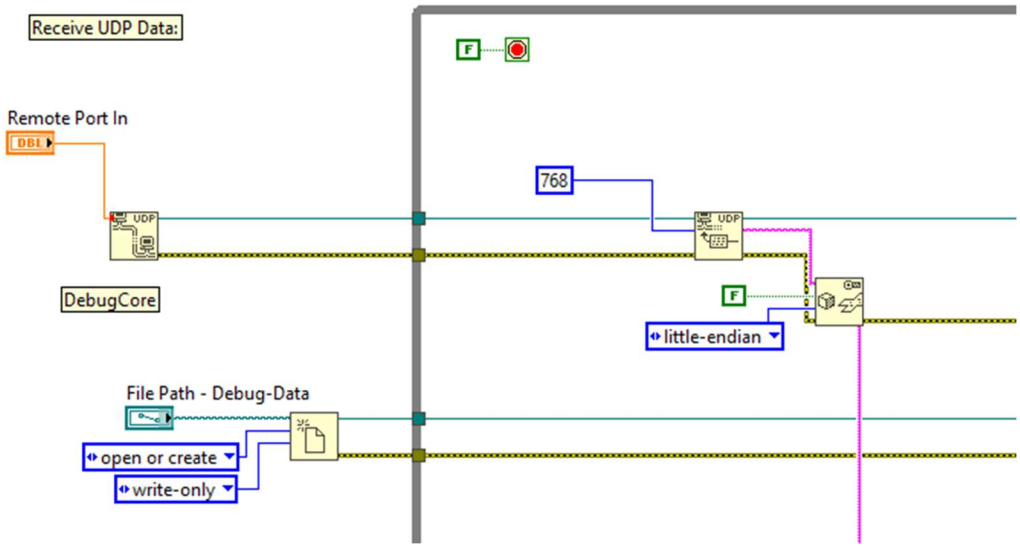


FIGURE 102: UDP RECEIVE

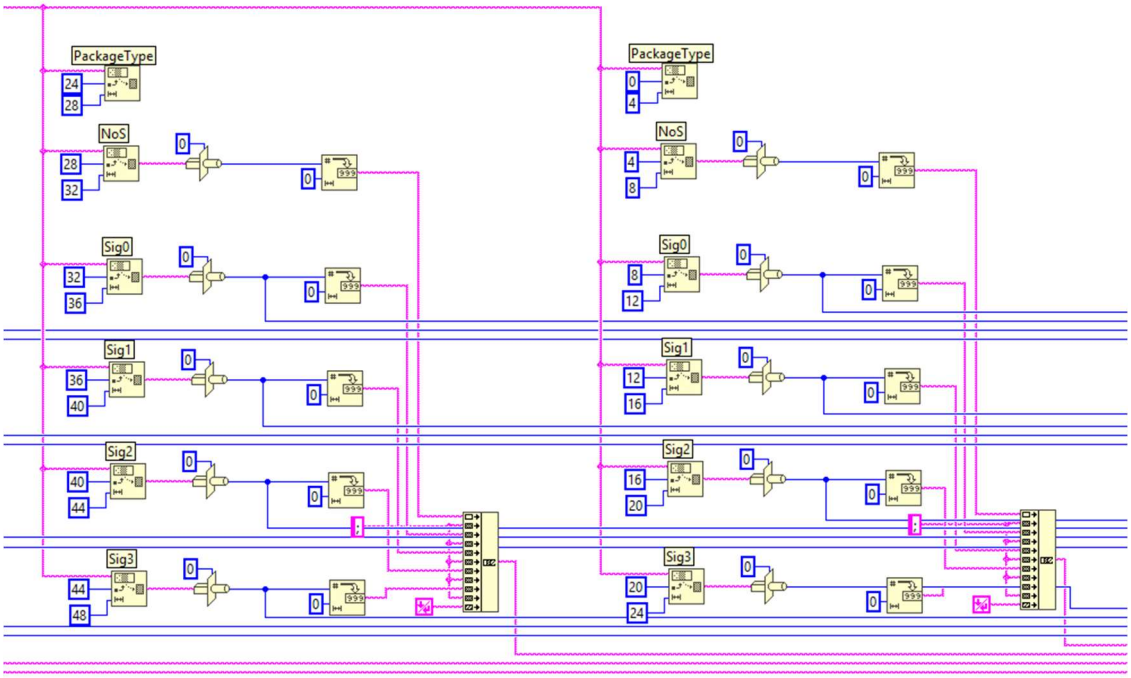


FIGURE 103: UDP DATA PROCESSING

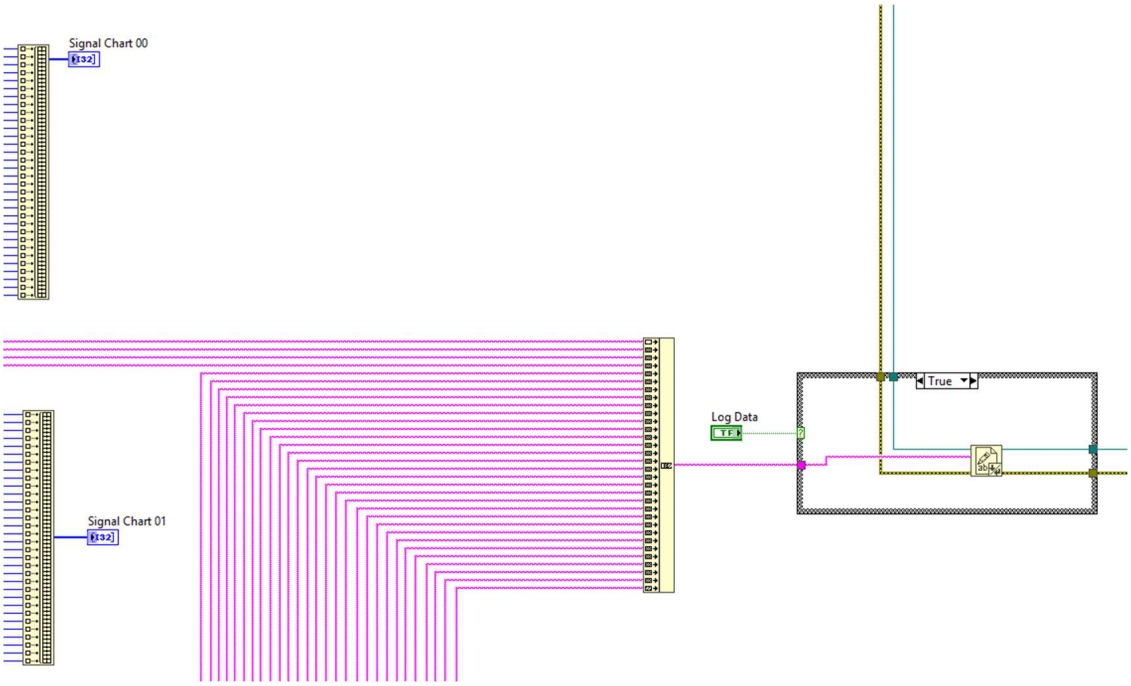


FIGURE 104: LOG DATA

9.3 PERFORMANCE OF THE LABVIEW USER-INTERFACE

The LabVIEW user-interface performance is bad. It works for low sample frequencies but at high sample frequencies the incoming UDP data are not processed very quickly. This causes data loss in the graphs and in the log files.

When the AID is debugging with a sample rate of 1 MHz, the receive buffer is full after a short time and LabVIEW cannot process the incoming UDP data as fast as they arrive. The receive buffer of the workstation is full and it overflows. Samples get lost and in the worst case, several thousand samples are lost. This leads to gaps in the graphs and in the log files. With lower sample rates, the performance becomes better.

The performance can be increased, when data logging is disabled. File operations are very slow and by disabling them, the operations inside the receiving and data processing while loop can be processed faster. This leads to a small increase of the performance but it is still not good enough to process bigger sample numbers with higher sample rates.

Another problem is checking the received package type. There are different packages, which are sent to the user-interface. The data package and the version number package. Both have a different package type. By checking the received package type, the performance gets even worse than with enabled data logging. Due to this behavior, the package check was removed in the final version of the LabVIEW user-interface.

There is a LabVIEW version available, which supports real-time operations and event triggered while loops, which executes with a frequency of 1 MHz, but a real-time operating system is necessary. Notebooks, which will be used for debugging do not have a real-time operating system. Therefore, this option is not feasible.

To increase the performance, parallelism is used. This was done by using different functions in multiple while loops. Functions can be entries or changes from the user-interface or data processing. These while loops work parallel to each other. The idea was, to slow down different while loops, with a Wait(ms) statement to increase the execution time of the while loops, which must run faster. Every function, which works with updates from the user-interface, was slowed down by 50ms. This effected the overall

performance with higher sample rates. The sample rates of 500 kHz and 250 kHz perform better. However, there is still a loss of data.

Handling the files is also difficult. There is a function to write to csv files, but this function performs every time an open file, write to file, and close file operation. This slows down the performance. This problem was reduced, by using an open file operation at the beginning of the receiving process. The data is logged with a write to file function. During the whole data logging process, the file stays open.

The problem with this solution is, that changing the filename, closing the file and other file dependent operations are difficult to do, when the file is still open. The csv file can only hold 1048576 [9] lines to open it in Microsoft Excel. It was not possible to change the file during the receiving process. Therefore, it is recommended to use the LabVIEW user-interface not in infinity mode.

Without the Wait statements in the while loops, the first data loss appears after nearly 2800 samples (87 UDP packages), at a sample rate of 1MHz.

Due to the parallelization of the while loops and slowing down the execution of some while loops, the performance was increased.

The user-interface was tested with the AID300 IP core and the ZedBoard. The test signals were generated by the SigGen_300 IP core. Figure 105 and Figure 106 show a debugging test with 7168 samples and a sample rate of 1 MHz. In Figure 105, the signals are monitored and there is no sample missing. However, when data logging is active, the performance drops due to the file operations and data loss occurs. This is shown in Figure 106.

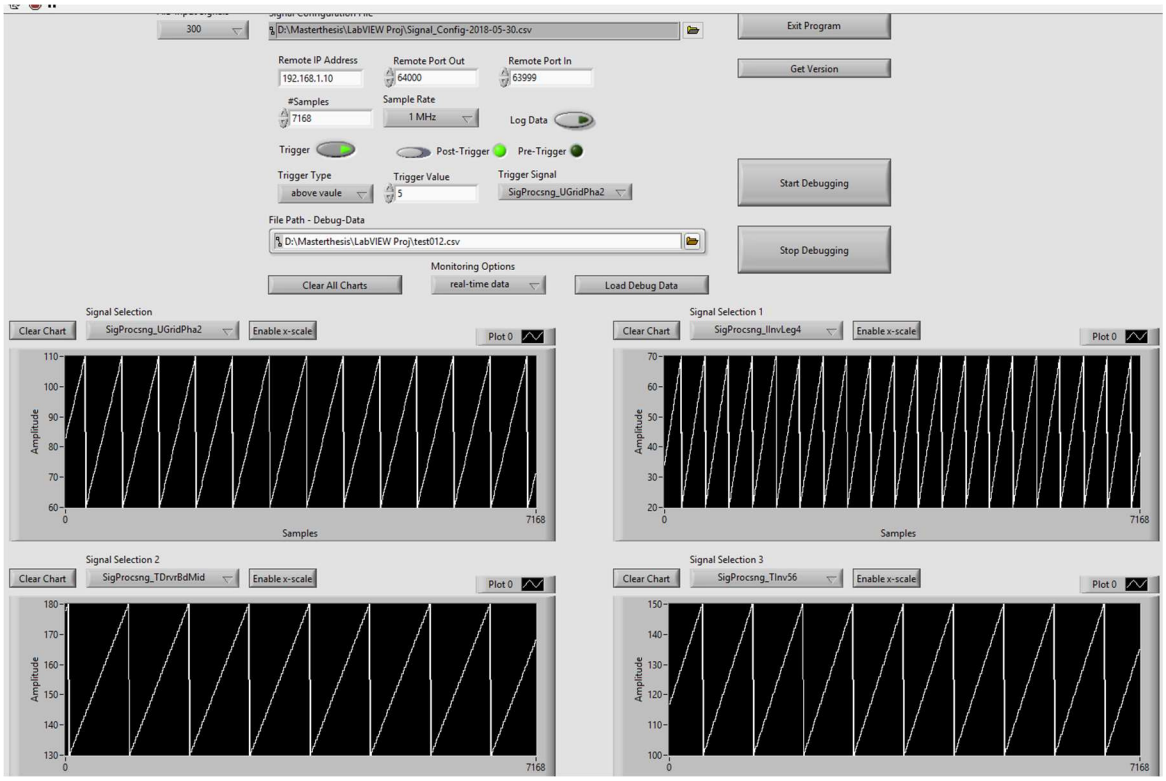


FIGURE 105: DEBUGGING TEST WITH 7168 SAMPLES AND A SAMPLE RATE OF 1 MHZ

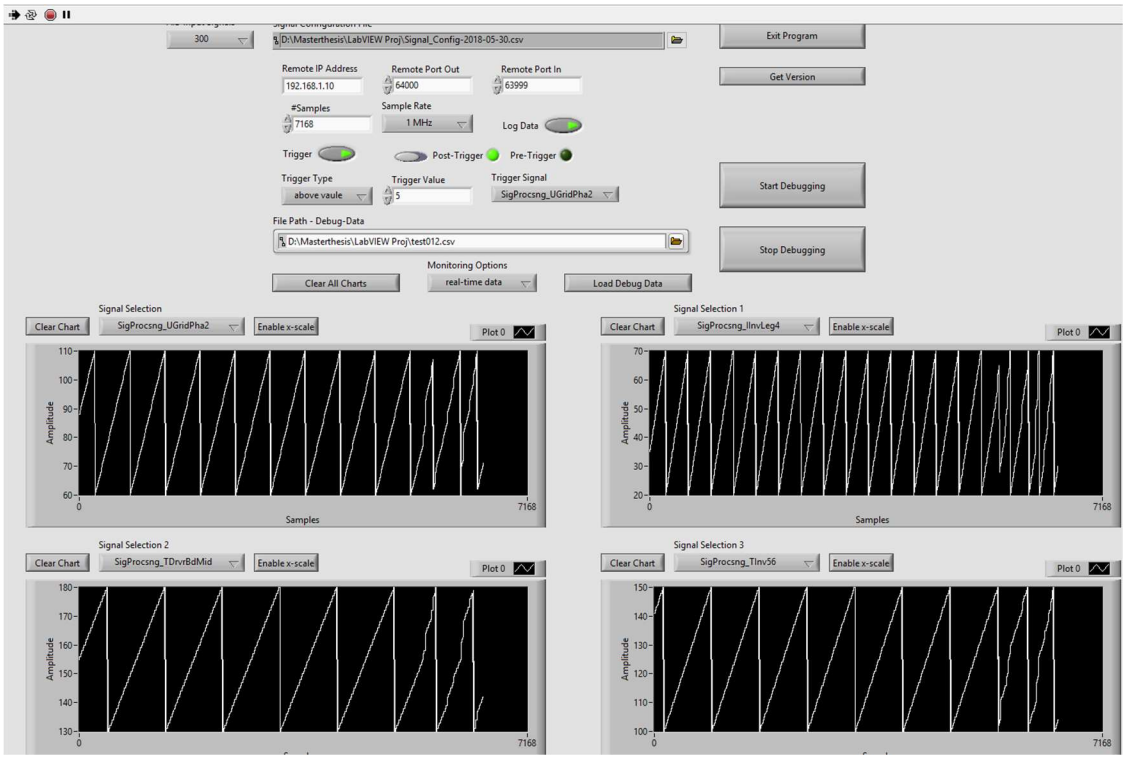


FIGURE 106: DEBUGGING TEST WITH 7168 SAMPLES AND A SAMPLE RATE OF 1 MHZ AND DATA LOGGING

Figure 107 shows the debugging process of 14336 samples with a sample rate of 500 kHz and data logging. Before reaching the final samples, glitches in the signal charts appear. These glitches are caused by the fast incoming UDP packages and the slow data processing. The receive buffer gets full and samples get lost. Not even with the modifications 14336 samples can be debugged correctly with a sample rate of 500 kHz when data logging is active.

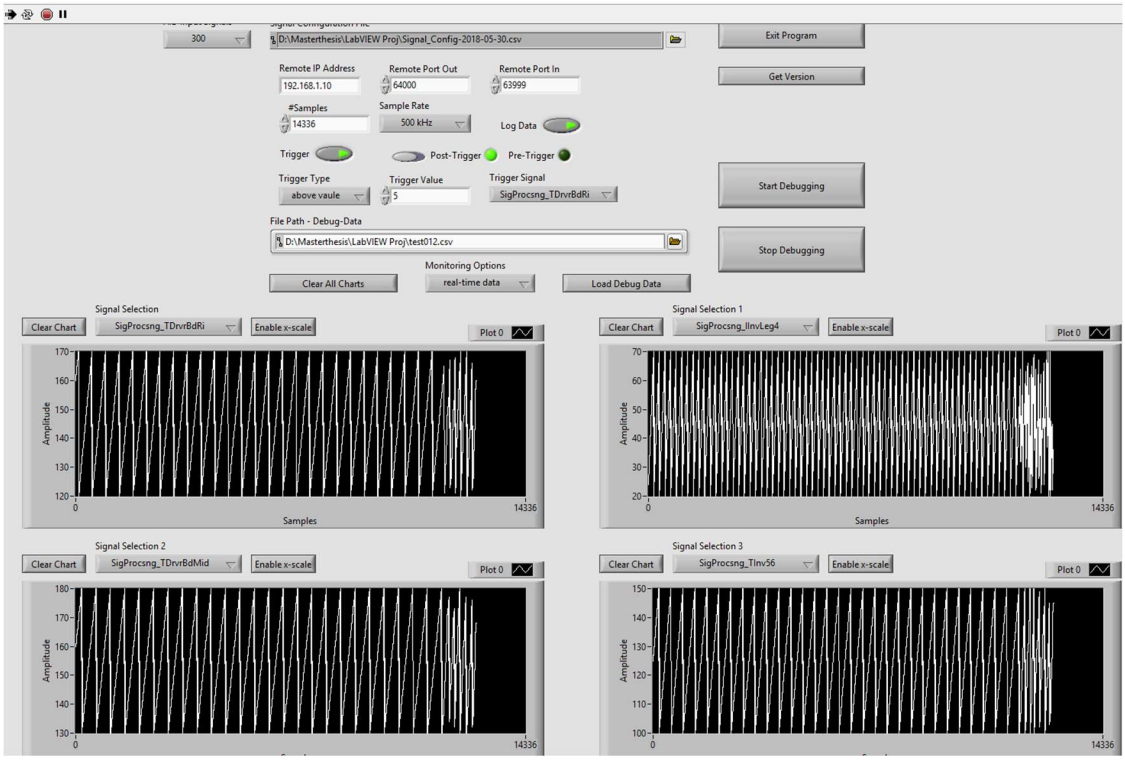


FIGURE 107: DEBUGGING TEST WITH 14336 SAMPLES AND A SAMPLE RATE OF 500 KHZ AND DATA LOGGING

Figure 108 shows a debugging test with the same settings but without data logging. It is possible to debug 14336 samples with a sample frequency of 500 kHz.

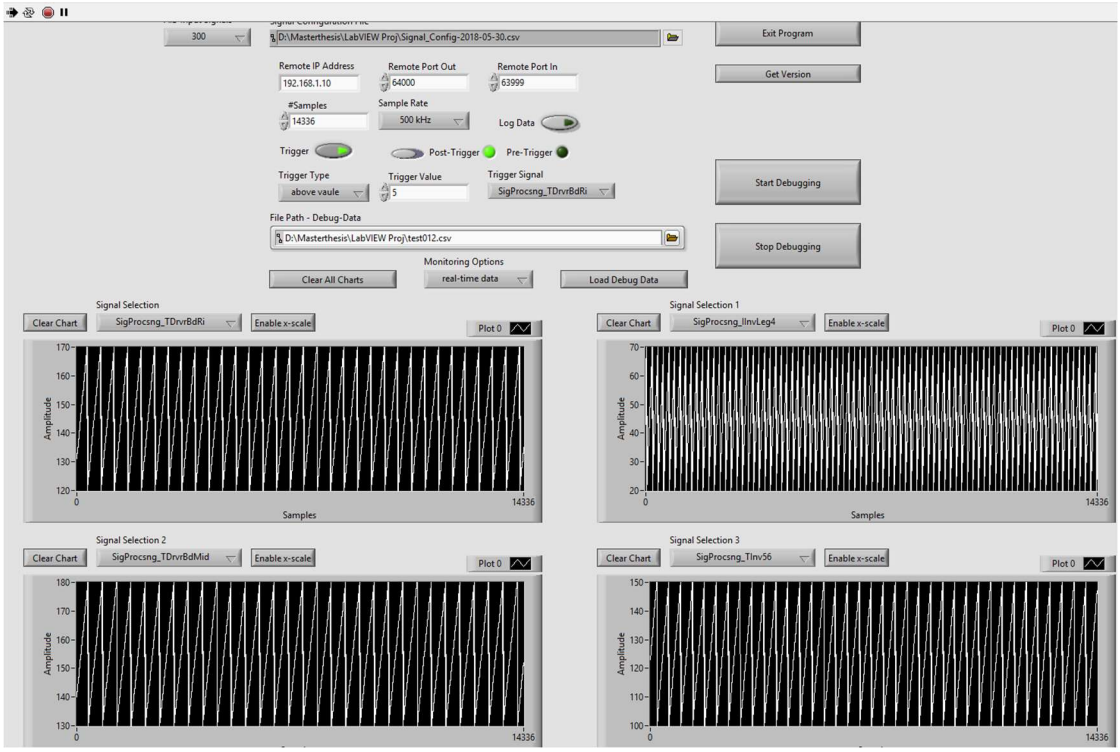


FIGURE 108: DEBUGGING TEST WITH 14336 SAMPLES AND A SAMPLE RATE OF 500 KHZ

Figure 109 shows the debugging process of 21504 samples with a sample rate of 250 kHz. There is no sample missing.

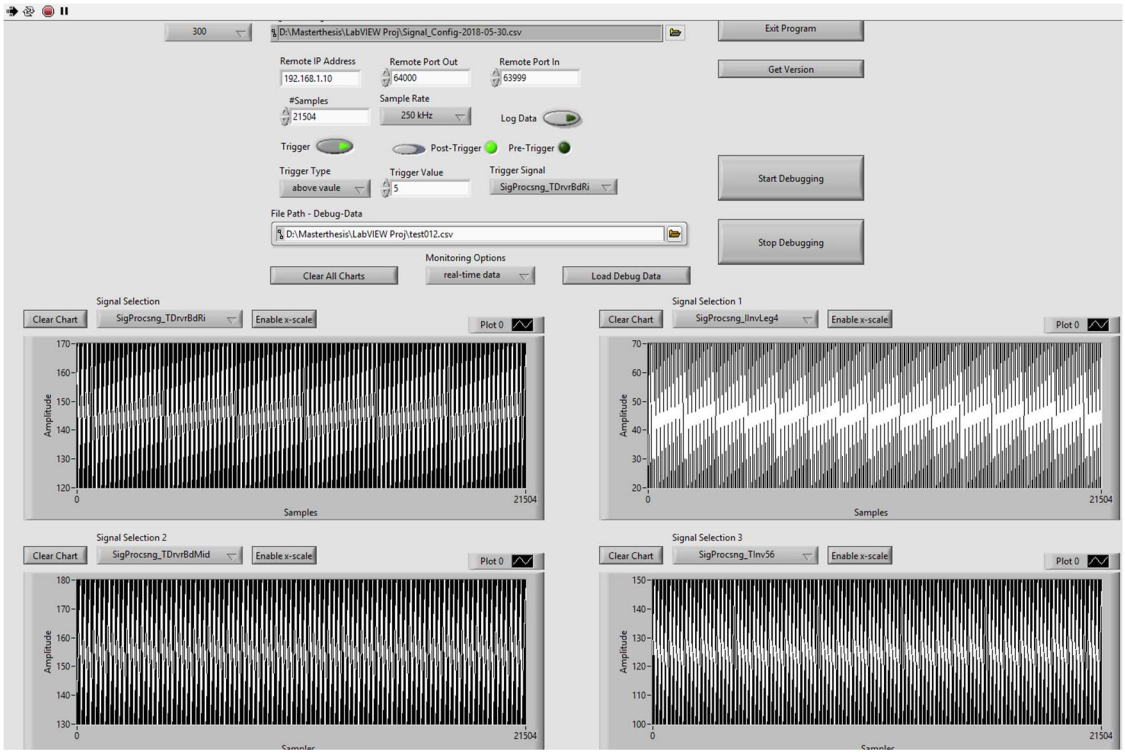


FIGURE 109: DEBUGGING TEST WITH 21504 SAMPLES AND A SAMPLE RATE OF 250 KHZ

Figure 110 shows the debugging process with the same settings but with data logging. It is still possible to debug 21504 samples with a sample rate of 250 kHz and log the data.

With lower sample rates, a higher number of samples can be adjusted for the debugging process.

It is still possible to debug 500736 samples with a sample rate of 250 kHz and enabled data logging. This is show in Figure 111.

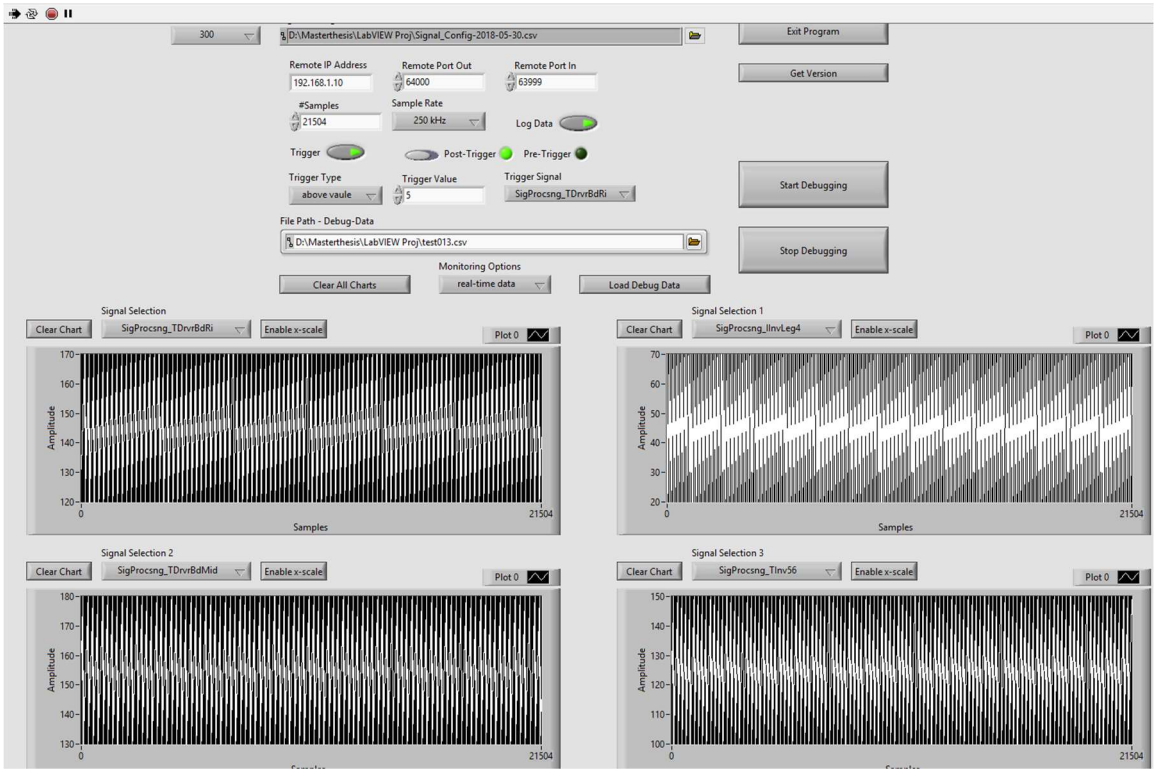


FIGURE 110: DEBUGGING TEST WITH 21504 SAMPLES AND A SAMPLE RATE OF 250 KHZ AND DATA LOGGING

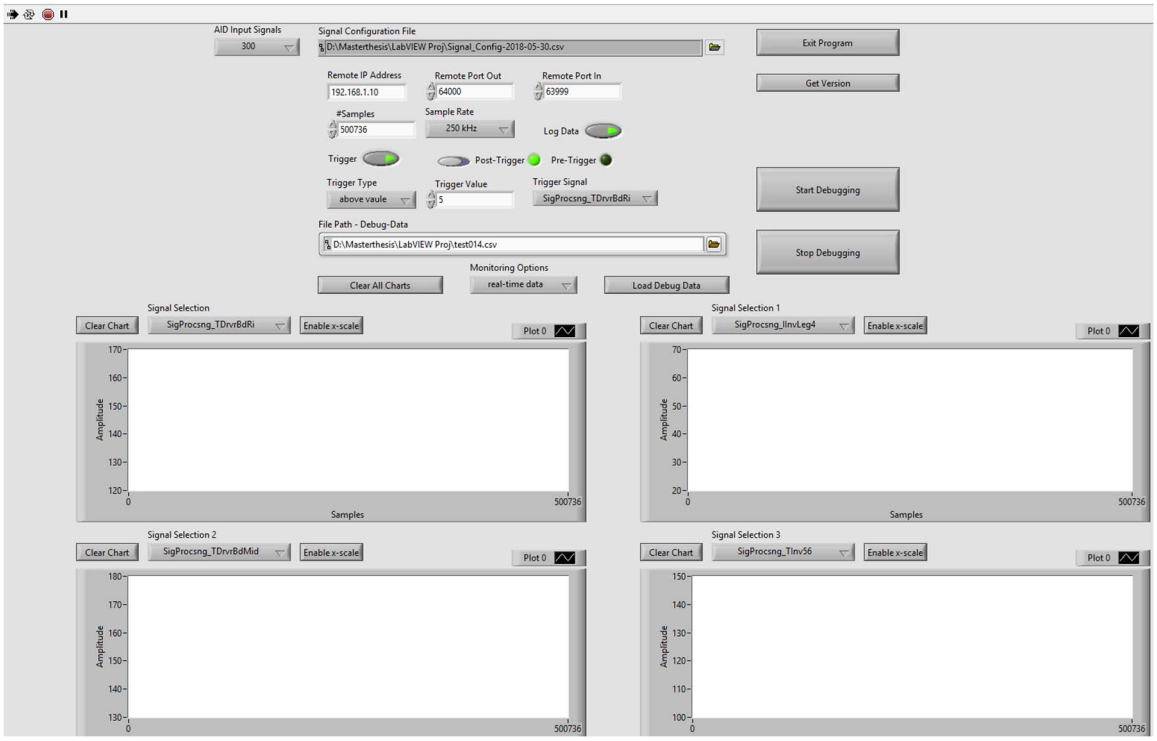


FIGURE 111: DEBUGGING TEST WITH 500736 SAMPLES AND A SAMPLE RATE OF 250 KHZ AND DATA LOGGING

Figure 112 shows the debugging process with infinity mode. The debugging is as long active, until the stop button is pressed. It is theoretical possible to debug as long as possible. In practice, it is different, due to the limitations explained before. After approximately 800000 received samples, the glitches occur. The data processing is too slow, the receive buffer overflows and data is lost.

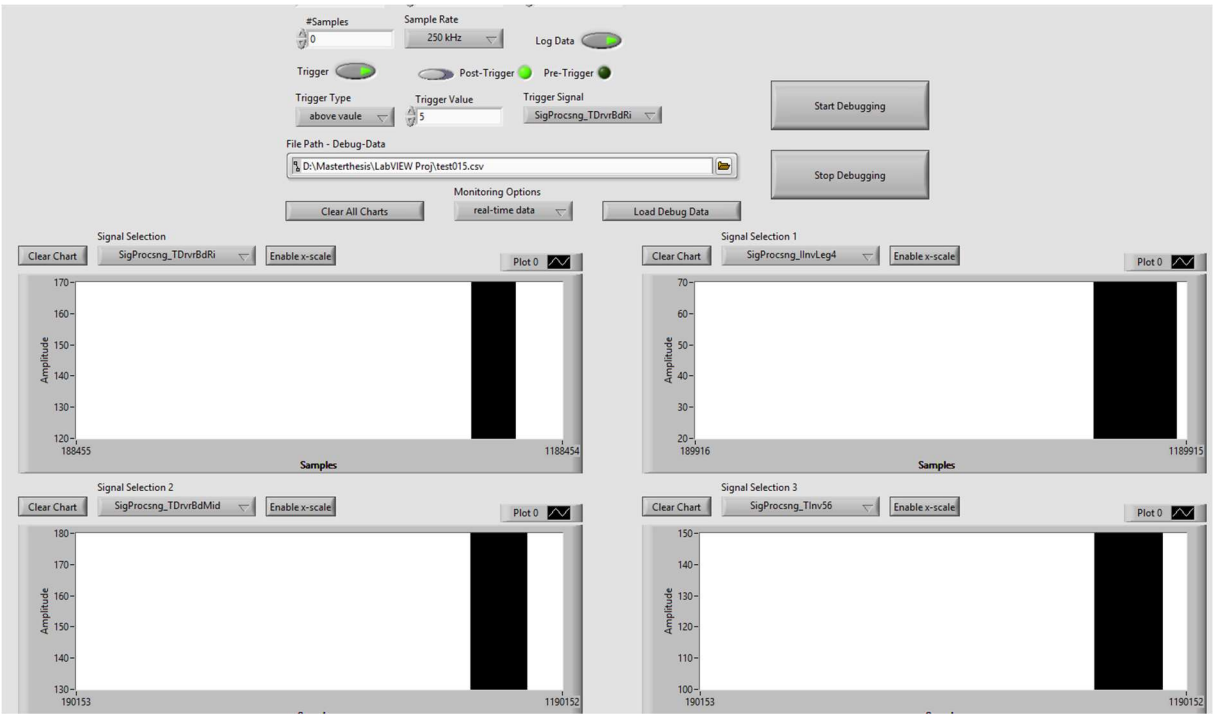


FIGURE 112: DEBUGGING TEST WITH INFINITY MODE WITH A SAMPLE RATE OF 250KHZ AND DATA LOGGING

10 C# USER-INTERFACE

To solve some problems with the LabVIEW user-interface, a new user-interface with C# and Microsoft Visual Studio[38] was developed. Visual Studio provides all the necessary libraries for this application. To create a user-interface, which looks like the LabVIEW user-interface, a C# Form application was created. The C# user-interface is better than the LabVIEW user-interface in several points:

- It was possible to include the version number into the GUI.
- Data logging in infinity mode can be done. The C# application automatically creates a new csv file with the appendix *-partxx.csv.
- The performance is better than the LabVIEW user-interface, due to multi-threading.
- It is possible to show the working status of the application.
- The zoom in the graphs is better handled.

[38] Microsoft Visual Studio, https://visualstudio.microsoft.com/

10.1 GUI

The C# user-interface controls the Advanced Inverter Debugger on the FPGA. After the adjustments for the desired debugging process, the C# application transmits the control information via a UDP connection to the target hardware and the FPGA. The application is called AID-UI, Advanced Inverter Debugger User-Interface and is shown in Figure 113.

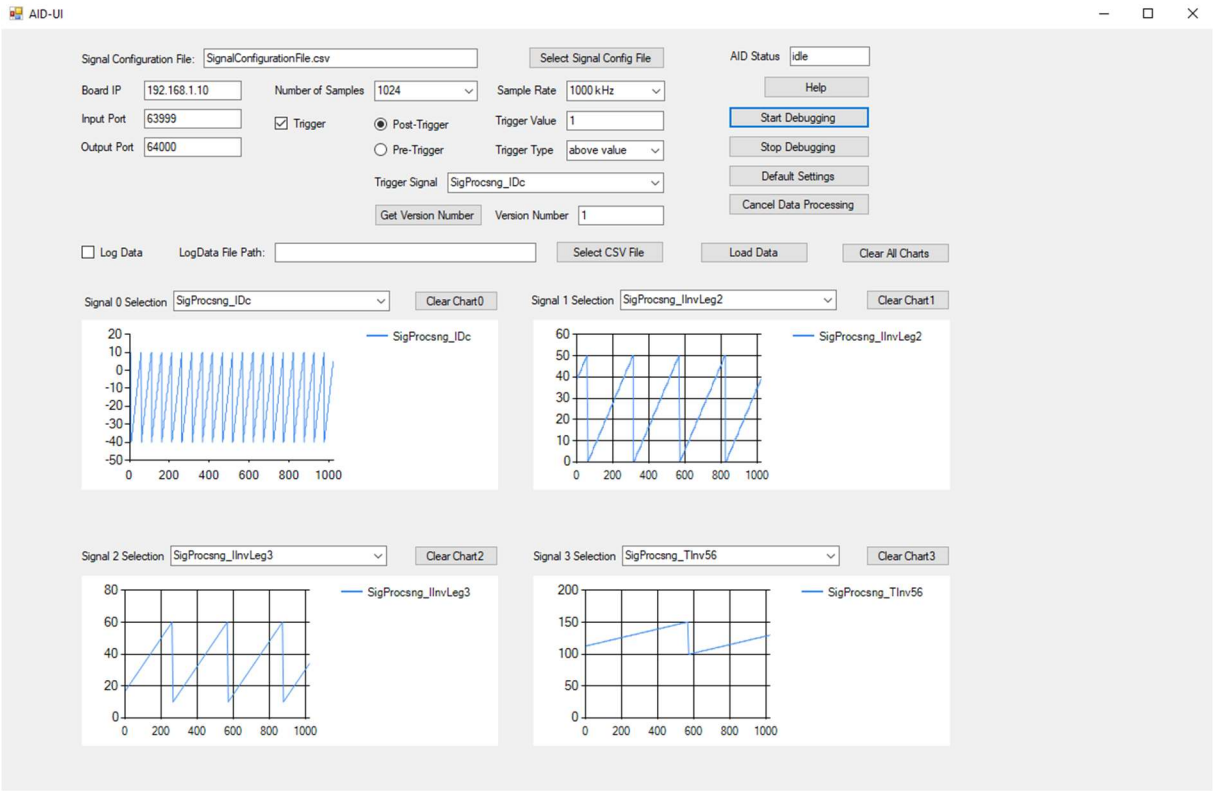


FIGURE 113: AID C# USER-INTERFACE

The C# user-interface is structured into different sections, like loading the signal configuration file, settings for the UDP connection, sampling settings, trigger settings, settings for data logging and the charts for monitoring the debugged data.

With the Select Signal Config File button, a file selection menu pops up to select the signal configuration file. The application checks if the file is a csv file and reads the data from the file. The trigger and signal selection combo boxes are updated with the signal names. If the file is no csv file, a pop-up window with the corresponding error message occurs.

When the AID-UI application is started, it tries to load a default signal configuration file. This default configuration file with the name, SignalConfigurationFile.csv, must be located in the directory, where the AID_UI.exe file is located. The trigger and signal selection combo boxes are automatically filled with the signal names of the default signal configuration file.

This feature is only working, if the file name and the file location is correct, otherwise the trigger and signal selection combo boxes stay empty and the signal configuration file must be selected with the Select Signal Config File button.

For the UDP connection between the C# user-interface and the FPGA, UDP connection settings are necessary. The settings contain the board IP address of the target HW, where the FPGA with the AID IP core is located and the port numbers for incoming and outgoing UDP packages. At the application

start, the default values are loaded. The default IP address is 192.168.1.10 (which is also adjusted at the target HW). The default input port is 63999 and the default output port is 64000.

The values can be changed in the user-interface. The IP address and the port numbers are checked for valid input, when the Start Debugging button is pressed. Port numbers between 49152 and 65535 are valid. These port numbers are dynamic or private ports [10] and are not registered. The string from the IP address text box is parsed to an IP address. If this parsing fails, an error message occurs. An error message will also occur, when the port numbers are invalid.

The sample settings contain the sample rate and the number of samples. The number of samples defines how many samples are debugged. The number of samples is a multiple of 1024. The minimum is 1024 and the maximum is 999424.

There are two operation modes for the AID IP core. The normal mode and the infinity mode. In the normal mode, the AID automatically stops debugging, when the adjusted number of samples is reached. In the infinity mode, the AID only stops, when the Stop Debugging button is pressed, otherwise it run infinitely long. To activate the infinity mode, **inf** (infinity mode) must be selected in the Number of Samples combo box.

The sample rate defines the sample frequency of the debugging process. There are 25 different sample frequencies to choose for the debugging process. The maximum sample frequency is 1 MHz and the minimum sample frequency is 1 kHz.

With the trigger settings, the trigger can be enabled. Pre- and post-trigger can be selected with the radio buttons. The trigger event is defined by the trigger type and the trigger value. Each trigger type is available for the post- and pre-trigger.

The trigger types are:

- Above trigger value
- Lower trigger value
- Equal to trigger value

If the trigger is active, the signal selection for signal 0 works with the Trigger Signal combo box. The Signal 0 Selection combo box is disabled to avoid conflicts. It is automatically updated with the selected signal from the Trigger Signal combo box. The chosen trigger signal is monitored in chart 0. Chart 0 always monitors the trigger signal, when a trigger is selected. If no trigger is enabled, the signal selection must be done with the Signal 0 Selection combo box.

To log the debugged data, it is necessary to check the Log Data check box. This activates the data logging. By pressing the Select CSV File button, a window appears to select the csv file. The selected file is checked for the correct ending (csv file). If the file type is invalid, an error message occurs. With the csv file format, it is possible to open the files with a normal editor, like Notepad++. It is also possible to load the csv files with Microsoft Excel. There is a limitation of 1048576 [9] lines to monitor the data.

When data logging is active and the AID is running in infinity mode, more than 1 million lines of data are logged. Each line contains a sample with the 4 signal values and the sample number. Microsoft Excel can only monitor 1048576 [9] lines of the csv file. To log more lines, the received UDP packages are counted. When the counter reaches the maximum number of samples for one file, a new file is created. The maximum number of samples per file is set to 999424 samples. The new file is named with the actual file name and additionally with the extension of –partX.csv, where X is the file counter. Every log file contains the header with the debugging settings. The logged data files with the extended file names are shown in Figure 114. In this case, the maximum samples were limited to 4096 per file, to test the creation of new files.

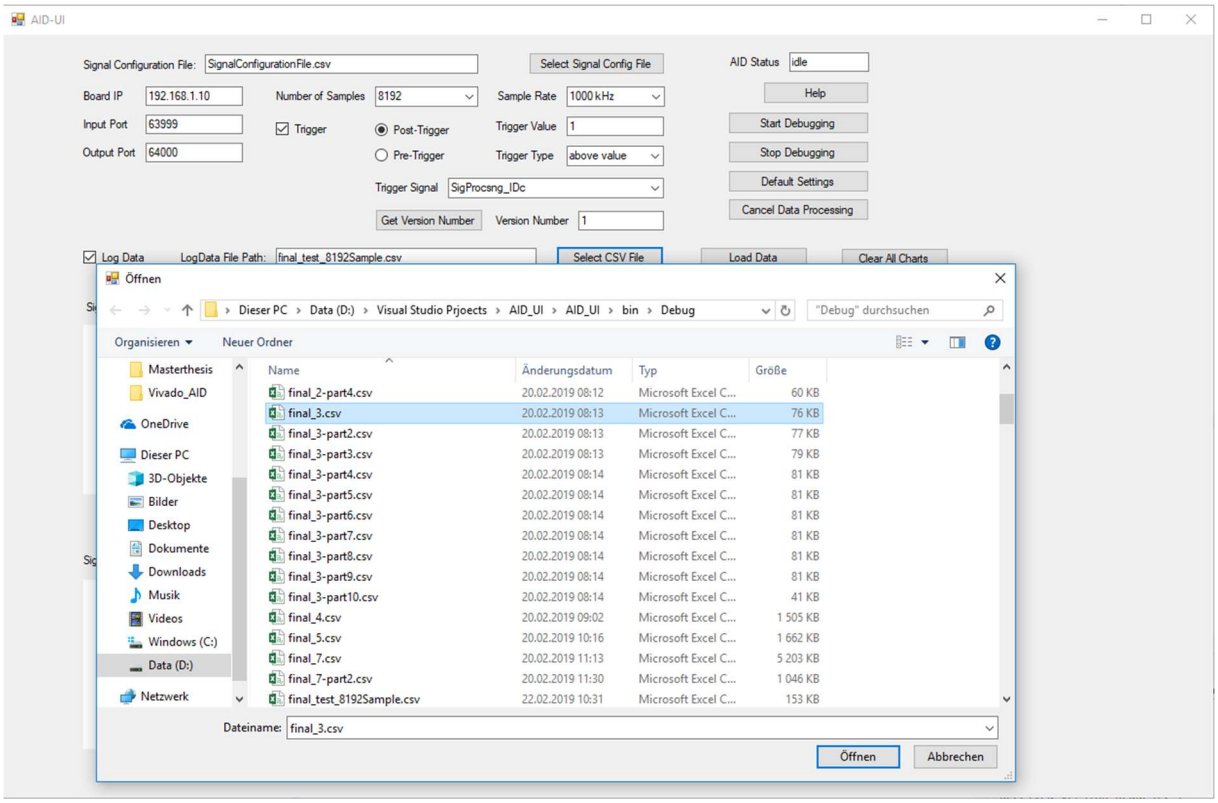


FIGURE 114: LOAD LOG FILES INTO THE USER-INTERFACE

When the Load Data button is pressed, the event handler creates the read data thread, which loads the data from the selected csv file into the charts. When the debugged data is logged into more than one file, the first log file must be selected. The user-interface is updated with the debugging information from the log files and the signal data of the other files is automatically loaded one by one into the charts.

The charts monitor the debugged signals. Each chart has its own signal selection combo box, to determine the signals for the debugging process. The charts are scalable by marking the desired window. Each chart has its own scale and a Clear Chart button. The Clear Chart button resets the chart history. There is also a Clear All Charts button, which resets the history of all charts.

To get the actual version number of the AID IP core, the Get Version Number button must be pressed. The button event creates a task, which handles the following steps: A new UDP client is created, which sends the control information with the package type 5 to the target HW. The Processing System with the UDP echo server receives the control information and due to package type 5, it responds with the version number of the AID. The version number is set in the driver files. The UDP package with package type 6 and the version number is sent to the AID-UI. The task receives the version number and monitors it in the Version Number text box.

The Start Debugging button, sends the control information to the AID IP core to start the debugging process. First, the different inputs from the user-interface are checked. If every input is valid, the UDP payload with the control information is created. Then, a UDP client is created to send the control information to the AID IP core. To receive and process the UDP packages, 2 threads are used. The first thread, t_recv, is the thread to receive all the incoming UDP packages. They are saved as byte arrays into a FIFO queue. The second thread, t_proc, is the thread to process the data from the FIFO queue. The data queue needs to be locked, to avoid concurrent access and to synchronize the threads. This is done by using a lock. The processing thread reads the data from the queue and converts the byte array into integer values and writes the data into the file, when data logging is active. It also updates the charts with the processed signal values. The thread also handles the maximum number of samples per file. When this number is reached, a new file is created. The receive thread runs with the highest priority, to

receive the UDP packages as fast as possible. The processing thread runs with normal priority. Both are running in the background, to avoid a frozen main window of the user-interface. The threads terminate themselves, when all received data has been processed.

The receive thread can also be terminated by pressing the Stop Debugging button. It creates a new UDP client, which sends the stop debugging control information to the AID IP core with the package type 3. This package resets the AID IP core on the FPGA.

If the FIFO queue is still full with received data, the processing thread runs as long as the data queue is full. When the queue is empty, a timeout occurs and the processing thread terminates itself. The processing thread can also be terminated by pressing the Cancel Data Processing button. This immediately terminates the processing thread and the FIFO data queue is cleared.

The Default Settings button, loads the default settings, like the UDP settings, which were discussed earlier.

The AID status defines the status of the user-interface. It can be in idle mode or in running mode. In idle mode, the AID-UI is able to perform every functionality. In running mode, the AID-UI is working in the background to process data. This can be converting the byte array from the data queue to integer values, updating the charts and writing the data to files or reading the data from the log files. Loading log files cannot be executed, when the AID-UI is in running mode.

The Help button activates the help window. It gives an overview of the different options of the AID-UI. The help window shows instructions to start the debugging process.

10.2 C# CLASSES OF THE USER-INTERFACE

The class Program.cs instantiates the AID-UI window and runs it.

The class Form1 is the main window with the name AID-UI. Form1 contains all the different elements, which are present in the user-interface. The Form1.cs handles all the different events, like pressed buttons. It instantiates the FileIO, UIHandler, UDP_Connection, UIStartConfiguration and AID_Thread. This is shown in the code below. These instances are used for the proper work of the AID-UI. The help window is created, when the Help button is pressed.

```
/// create instances
uiHandler = new UIHandler();
udp_con = new UDP_Connection();
ui_startconfig = new UIStartConfiguration();
csvFileIO = new FileIO();
t_aid = new AID_Thread();
```

The class Form2 is the help window. It contains only a text box, which is read only. The text is assigned during the initialization phase.

The class FileIO handles the file access. It handles the read operation from the signal configuration file. Therefore, the file path is checked for validity and the configuration data is loaded into the user-interface. It also handles the file access to log and load the signal data. Therefore, the file path is checked for validity, the header and the signal data is written into the log file. The header gives information about the adjusted parameters of the debugging process. It also automatically creates new log files with the extension –partX.csv, when more than one log file is required. To read the log files, the file path is also checked and if more than one log file exists of the same debugging process, it automatically loads all of the created log files. Furthermore, the header information of the debugging process will be read.

The class AID_Thread contains the receive thread, for receiving the UDP packages, the processing thread, for processing the received UDP data and the read data thread, for loading the logged data into the charts. The receive thread fills the data queue with the received UDP data. The processing thread converts the UDP data into integer values, which are written into the files, if data logging is active and the charts are updated. The read data thread opens the log files and loads all the data into the charts. The user-interface settings are automatically updated with the header information of the csv files. The header of the log file contains the settings of the debugging process. The signal data is monitored with the charts.

The UDP_Connection class handles the UDP connection. It creates the UDP clients to send and receive UDP packages. It also builds the UDP payload with the adjusted settings from the user-interface and sends the control information to the AID IP core. This control information can either be a start debugging command with the necessary debugging parameters, a stop debugging command or a get version number request.

The class UIHandler handles all the inputs from the user-interface. It checks the different elements for validity and handles error cases. It also updates elements when the signal configuration file or log files are loaded.

The class UIStartConfiguration sets up the default values of the user-interface. It also fills the sample rate combo box, the number of samples combo box and the trigger type combo box with the corresponding values.

10.3 PERFORMANCE OF THE C# USER-INTERFACE

The performance of the C# user-interface is better than the performance of the LabVIEW user-interface. Whit the threads running in the background, 60000 samples can be debugged with a sample rate of 1 MHz. With lower sample rates, it is possible to debug even more samples, without gaps in the charts. The user-interface was tested with the ZedBoard as target hardware. The SigGen300 was used to generate the test signals for the AID300. Figure 115 shows the debugging process with 60416 samples and a sample rate of 1MHz.

The performance was increased. However, gaps can still appear. These gaps are caused, when the receiving thread is blocked by the processing thread. These gaps are usually very small. Due to the blocked receiving thread, time is wasted and at some point, gaps of missing samples will appear. Big gaps are caused, when the UDP packages are sent too fast. The receive buffer of the network card overflows and samples get lost.

Figure 116 shows the two different kinds of gaps in the charts. The charts for signal 0 and signal 2 have bigger gaps, with about 2000 missing samples. These gaps are caused by the full receive buffer. The receive buffer overflows and the samples get lost. The chart for signal 1 shows the smaller gap, which is caused, when the processing thread blocks the receiving thread.

The gaps can occur at lower sample rates as well, but in these cases, the gaps are small and about 2000 samples are lost. At higher sample rates, the gaps of missing samples can increase up to 20000 samples, depending on the adjusted number of samples for the debugging process. The worst case is a combination of a full receive buffer and a blocked receiving thread.

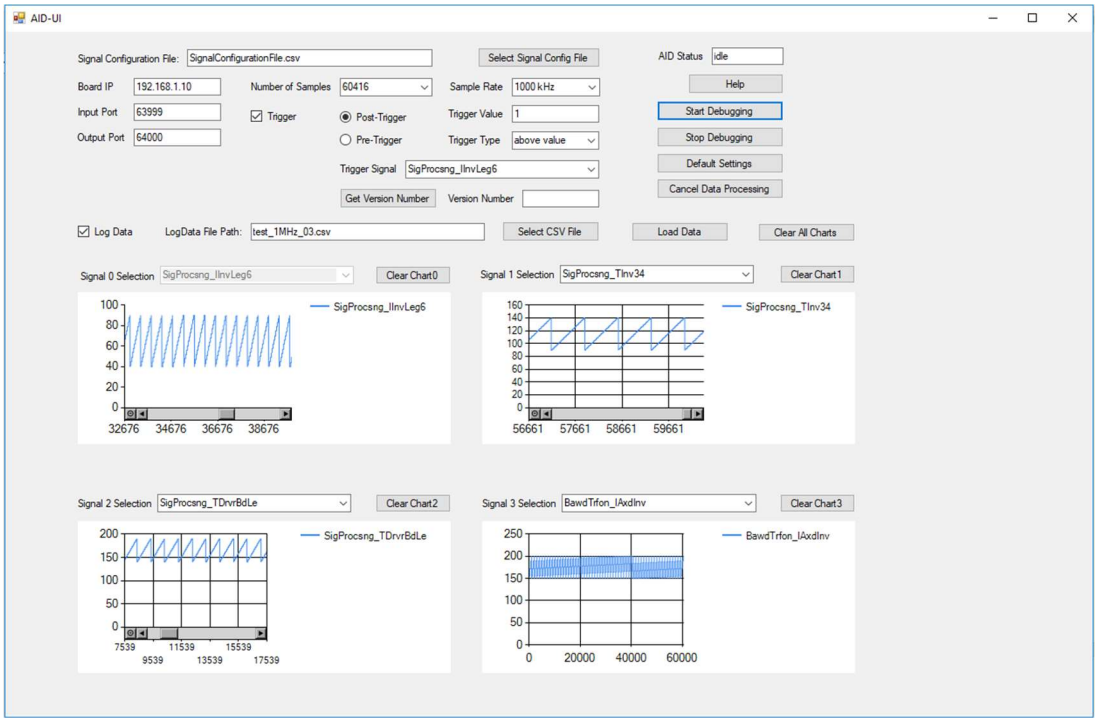


FIGURE 115: DEBUGGING PROCESS OF 60416 SAMPLES WITH A SAMPLE RATE OF 1MHZ

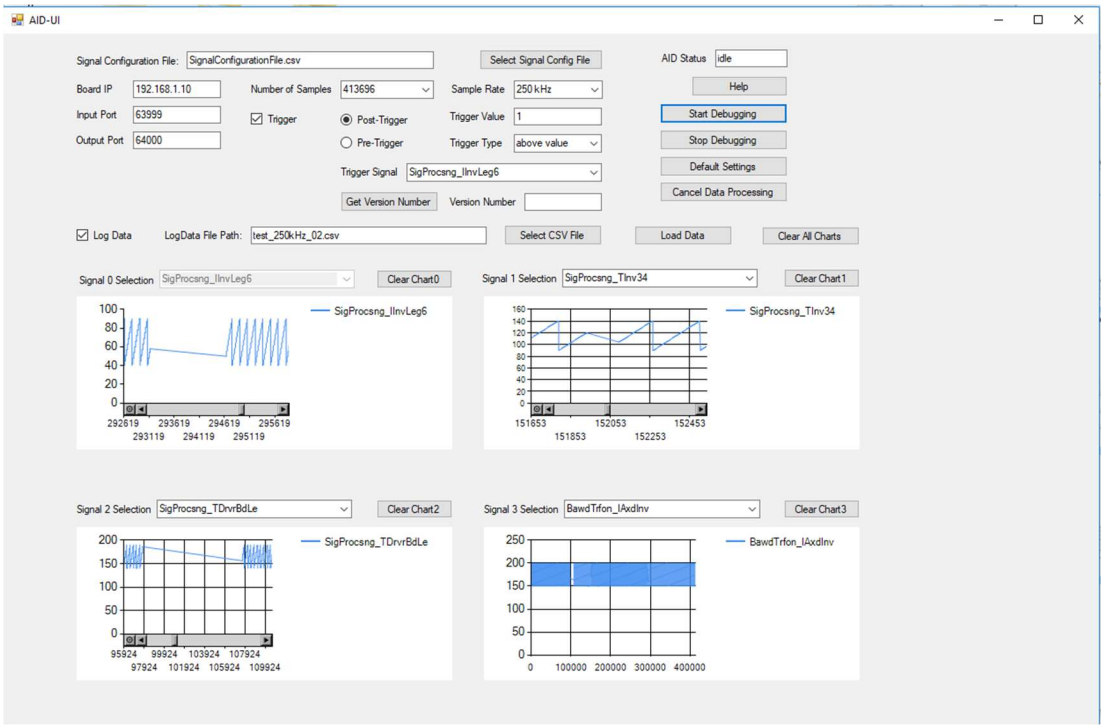


FIGURE 116: GAPS IN THE CHARTS AT A SAMPLE RATE OF 250 KHZ

Figure 117 shows the worst case of gaps. In that case, the sample rate is 500 kHz and the receive thread was blocked by the processing thread. The receive buffer overflows and data is lost. 15000 samples are lost. This behavior is worse with a sample rate of 1MHz. The higher the sample rate and the higher the number of samples, the more samples get lost.

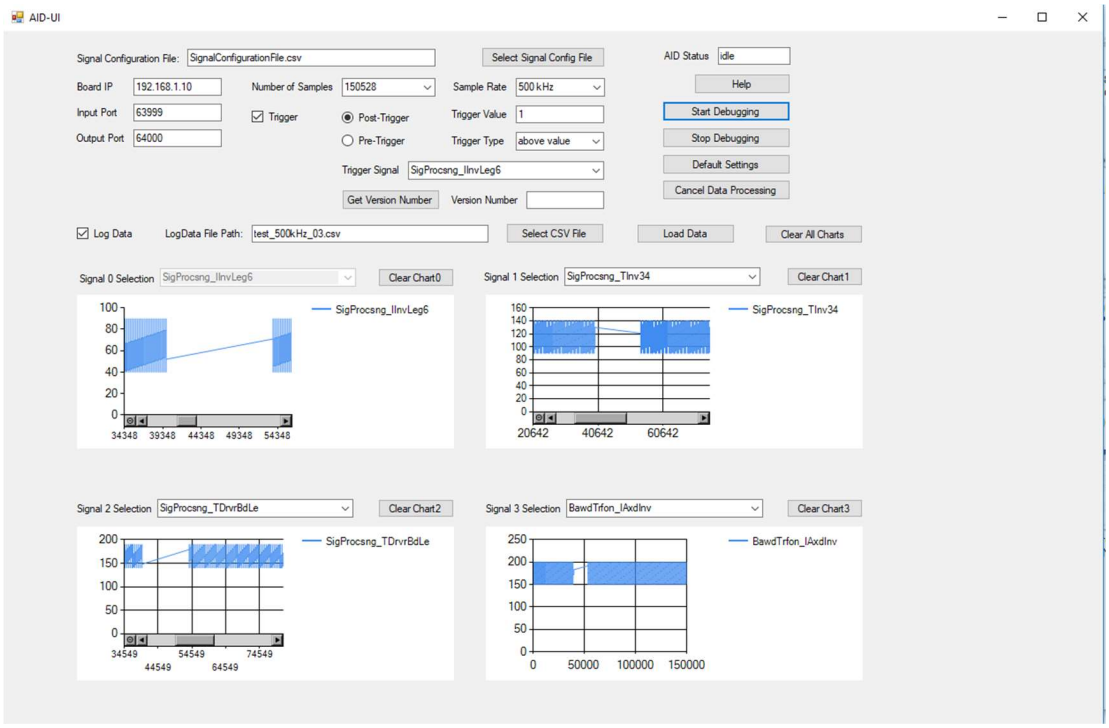


FIGURE 117: CHARTS WITH BIG GAPS AT A SAMPLE RATE OF 500 KHZ

Figure 118 shows the debugging process with a sample rate of 1MHz and a number of samples of 100352. Due to the blocked receiving thread and the fast sample rate, the data loss occurs earlier. There are also more lost samples because, the UDP packages are sent with a higher frequency.

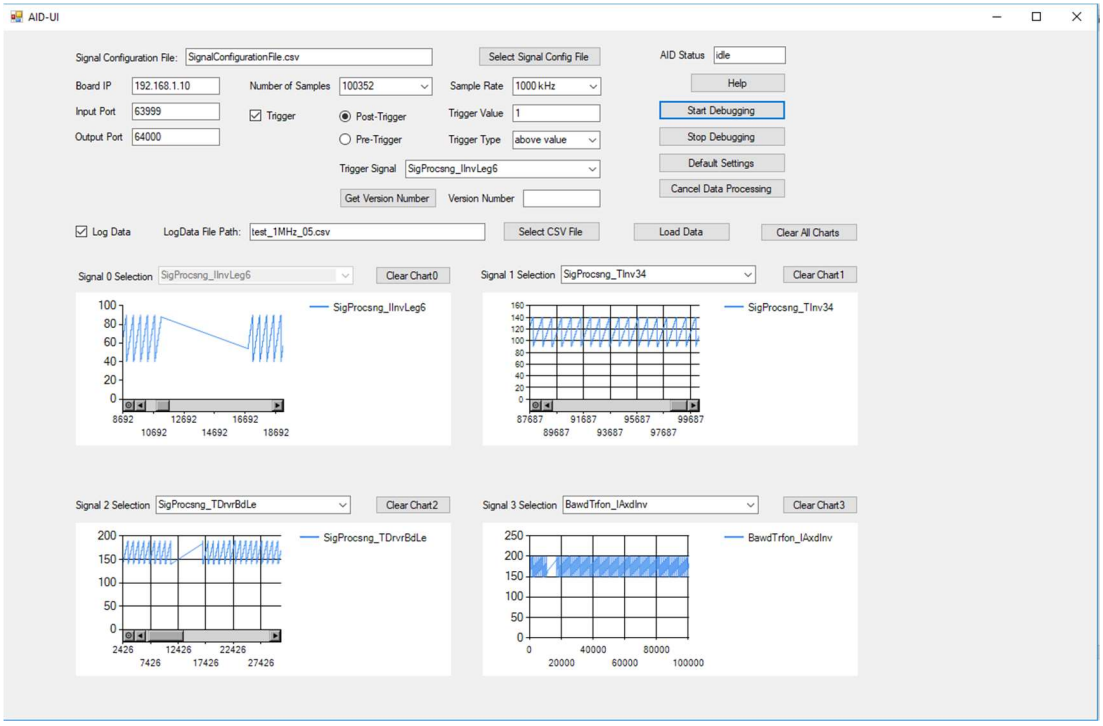


FIGURE 118: LOST SAMPLES AT SAMPLE RATE OF 1MHZ

The performance can still be increased by using different libraries for the charts. The update process for the charts takes too much time. Every time, the charts are updated, the Invoke function is called and the thread is changed to the main thread of the user-interface. The main thread is the window form application, which stays open, as long until the close button is pressed.

The charts update very slowly. With the charts from the C# library, it is not possible to update the charts by appending them with data arrays until the debugging process is done. Therefore, every point has to be added in a loop. To make it still faster, the update process of the charts in the main window is done every 1024 samples. The update code is shown below.

```
// add point to charts
cSig0.Invoke(new Action(() => { cSig0.Series[0].Points.AddXY(sample_number, sig0); }));
cSig1.Invoke(new Action(() => { cSig1.Series[0].Points.AddXY(sample_number, sig1); }));
cSig2.Invoke(new Action(() => { cSig2.Series[0].Points.AddXY(sample_number, sig2); }));
cSig3.Invoke(new Action(() => { cSig3.Series[0].Points.AddXY(sample_number, sig3); }));

// update charts
if ((receivedSamples % 1024) == 0)
{
        cSig0.Invoke(new Action(() => {
                cSig0.Series[0].Points.ResumeUpdates();
                cSig0.ChartAreas[0].AxisX.Maximum = sample_number;
                //cSig0.Series[0].Points.SuspendUpdates();
        }));
        cSig1.Invoke(new Action(() => {
                cSig1.Series[0].Points.ResumeUpdates();
                cSig1.ChartAreas[0].AxisX.Maximum = sample_number;
                //cSig1.Series[0].Points.SuspendUpdates();
        }));
        cSig2.Invoke(new Action(() => {
                cSig2.Series[0].Points.ResumeUpdates();
                cSig2.ChartAreas[0].AxisX.Maximum = sample_number;
                cSig2.Series[0].Points.SuspendUpdates();
        }));
        cSig3.Invoke(new Action(() => {
                cSig3.Series[0].Points.ResumeUpdates();
                cSig3.ChartAreas[0].AxisX.Maximum = sample_number;
                //cSig3.Series[0].Points.SuspendUpdates();
        }));
}
```

There are libraries for fast data monitoring available but they are not open source and you need a license to use them. That is why the standard C# library was used.

Due to UDP/IP, UDP packages can arrive in the wrong order. Figure 119 shows the entries of the log file. The second UDP package with the samples 33-64 are written into the file followed by the first UDP package with the samples 1-32. Figure 120 also shows the wrong order in the charts of the user-interface.

The performance of the C# user-interface also depends on the notebook or workstation. The threads can be processed faster, when the processors are faster and the cache/RAM is bigger.

	A	B	C	D	E	F	G
1	DebugCor	Date: Mitt	Time:		10:41:59		
2	Number o	Sample Ra	Trigger:		Pre-Trigge	Trigger Ty	Trigger Value:
3	1024	1000 kHz	TRUE	FALSE	above val	1	
4	Number o	SigProcsn	SigProcsn	SigProcsn	SigProcsng_IDc		
5	33	-14	-14	-14	-14		
6	34	-13	-13	-13	-13		
7	35	-12	-12	-12	-12		
8	36	-11	-11	-11	-11		
9	37	-10	-10	-10	-10		
10	38	-9	-9	-9	-9		
11	39	-8	-8	-8	-8		
12	40	-7	-7	-7	-7		
13	41	-6	-6	-6	-6		
14	42	-5	-5	-5	-5		
15	43	-4	-4	-4	-4		
16	44	-3	-3	-3	-3		
17	45	-2	-2	-2	-2		
18	46	-1	-1	-1	-1		
19	47	0	0	0	0		
20	48	1	1	1	1		
21	49	2	2	2	2		
22	50	3	3	3	3		
23	51	4	4	4	4		
24	52	5	5	5	5		
25	53	6	6	6	6		
26	54	7	7	7	7		
27	55	8	8	8	8		
28	56	9	9	9	9		
29	57	10	10	10	10		
30	58	-40	-40	-40	-40		
31	59	-39	-39	-39	-39		
32	60	-38	-38	-38	-38		
33	61	-37	-37	-37	-37		
34	62	-36	-36	-36	-36		
35	63	-35	-35	-35	-35		
36	64	-34	-34	-34	-34		
37	1	5	5	5	5		
38	2	6	6	6	6		
39	3	7	7	7	7		

wrongsampleorder

FIGURE 119: LOG FILE WITH WRONG SAMPLE ORDER

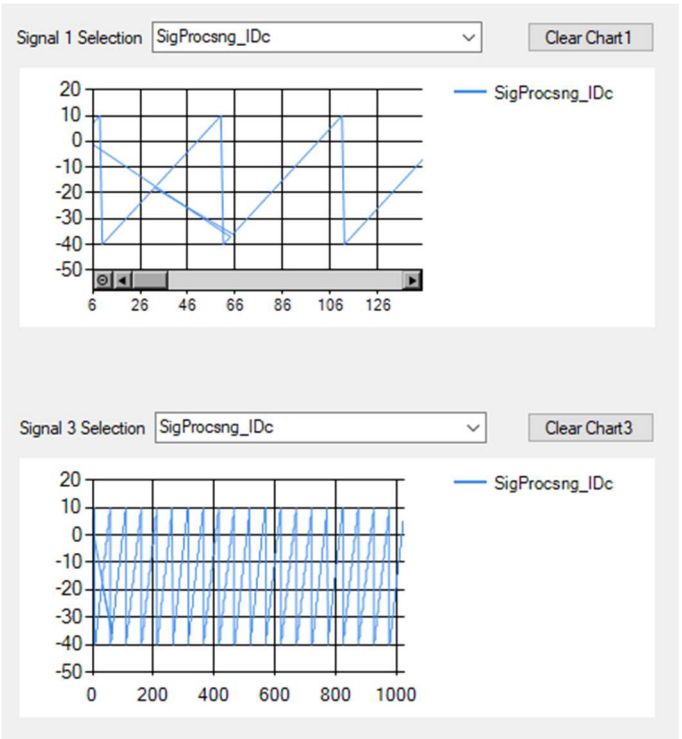


FIGURE 120: SAMPLES RECEIVED IN THE WRONG ORDER

11 UDP CONNECTION

The communication between the AID and the user-interface is done with UDP/IP. To accomplish the communication, an UDP echo server is running on the Processing System of the ARM-Processor. The server handles incoming and outgoing UDP packages.

The user-interface creates a UDP client, which establishes a connection to the server and the data is transmitted. There are different package types available. The UDP packages with the control information is structured in the same way to start and reset the AID and for the request of the version number. A typical payload from an UDP package from the user-interface to the AID is shown in Figure 121. The payload is structured in package type, number of samples, sample rate, CMD, trigger type, trigger value and 4 signals with the bit size of every entry.

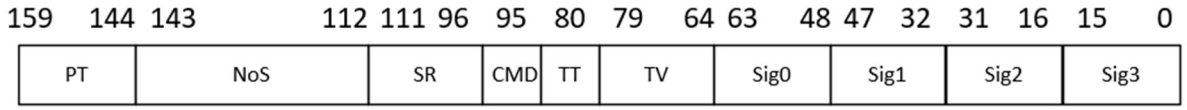


FIGURE 121: UDP PAYLOAD OF THE CONTROL INFORMATION

The package type defines, which package it is. The package types are discussed in the next section.

Figure 122 shows an overview of the command section. The MSB is the start bit. It enables the debugging process when the package type is Start Debugging. The reset bit, resets the AID to stop the debugging process when the package type is Stop Debugging. The trigger bit activates the Post-Trigger. To enable the Pre-Trigger, the Trigger bit and the Pre-Tr bit must be set. The trigger settings are only important with the Start Debugging package. The other bits are reserved for the future but currently not used.

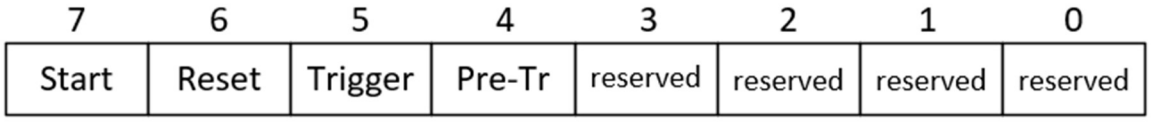


FIGURE 122: STRUCTURE OF THE COMMAND SECTION

Figure 123 shows an overview of the UDP payload of a data package. The payload is structured when every sample is sent directly to the user-interface. Due to the Processing System, this is not possible. The fastest possible way is, to collect at least two of these data packages for the UDP payload. In the current implementation, 32 of these data packages are stored in the RAM and then sent with the UDP package to the user-interface.

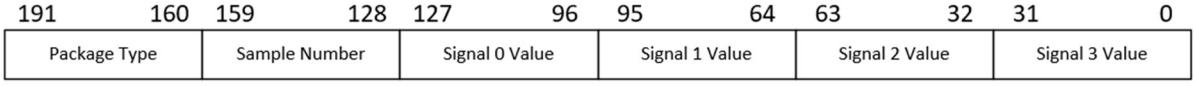


FIGURE 123: UDP PAYLOAD WITH ONE DATA PACKAGE

Figure 124 shows the UDP payload for the transfer of the version number. The package type defines the package as version number with its value. The version number request is checked by the Processing

System and also handled by the Processing System. It sends the package type with the version number back to the user-interface.

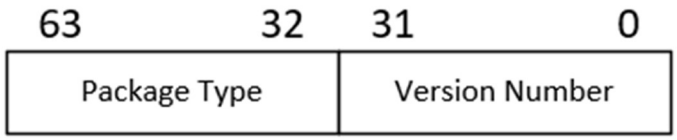


FIGURE 124: UDP PAYLOAD WITH VERSION NUMBER

11.1 PACKAGE TYPE NUMBERS

There are several UDP package types for the communication between the user-interface and the AID. The different package types with their numbers are shown in Table 5.

TABLE 5: PACKAGE TYPES

Package Type	Function
0	Start Debugging, contains the information to start a debugging process
1	Data Package, contains the information of 1 data sample. 32 of these data packages are currently collected for the UDP payload
3	Reset, contains the information to reset the Debug-Core
5	Get Version Number, contains the information for the version number request. This request is handled with the PS.
6	Version Number is the acknowledge to the version number request and contains the version number

The package type 0 is the Start Debugging package. Every information of the rest of the package can be adjusted with the user-interface. The control data contains number of samples, sample rate, trigger information, chosen signals and the command Byte. When the start bit of the command section is set, the AID will start the debugging process. The UDP payload size is 20 Bytes.

The package type 1 is the data package. It contains the sample number and the signal values. One data package is 24 Bytes long. Currently, 32 of these data packages are collected and sent with one UDP package to the user-interface. The UDP payload size is 768 Bytes.

The package type 3 is the reset package. It is used to reset the AID or to stop the AID from debugging. When the AID is running in infinity mode, the reset package is sent to stop the debugging process. The UDP payload is 20 Bytes long.

The package type 5 is the package for the version number request and is processed from the Processing System. The version number is sent back to the user-interface.

The package type 6, defines the package as version number. It contains the package type and the version number.

11.2 SETTINGS FOR THE PROCESSING SYSTEM

The UDP echo server from the Processing System of the ZedBoard or Controller Board is set up without using the checksum. This increases the performance of the PS. The settings for the UDP echo server are in the aid_settings.h file. The following code shows the settings for the echo server:

```
/*
 * increase speed of udp
 * 1. Remove section in code for tcp/ip
 * 2. Manually assigning MAC and IP -> not 15-20 sec bootup for DHCP
 * 3. Reduce overhead for checksum
 * 4. Reduce overhead for checksum
 * 5. Reduce overhead for checksum
 */
#undef LWIP_TCP
#undef LWIP_DHCP
//#undef CHECKSUM_CHECK_UDP
//#undef LWIP_CHECKSUM_ON_COPY
//#undef CHECKSUM_GEN_UDP
```

12 TESTING THE AID IP-CORE ON THE ZEDBOARD

The first tests were made with the ZedBoard. The inverters need more resources, than the ZedBoard provides. Therefore, the signal generator Sig_Gen300 is used to generate the test signals. The signal generator generates counter signals with different frequencies and constant values. There is a signal generator with 40 and 300 output signals available.

To test the design with the ZedBoard, the ZYNQ Processing System was added to the block design. It was modified with a High Performance AXI Slave Port and 2 Fabric Clock signals. The frequencies of the clocks are 1 MHz and 100 MHz. The 100MHz clock is used for the AXI IP cores and for the AID IP core. The 1MHz clock is used for the AID IP core and for the signal generator IP core.

For the communication the Ethernet and UART peripheral I/O pins are enabled. The UART pins and the Xilinx ILA core are used to verify the correct work of the AID IP core.

In the ZYNQ7 Processing System Settings, the PL-PS interrupts (Program Logic to Processing System interrupts) are enabled in the section interrupts. The access to the DDR memory is enabled with the High Performance AXI Slave ports in the section PS-PL Configuration (Processing System to Program Logic). This interface is used to write the received control information into the RAM. The data is read with the AXI DataMover to control the AID IP core. On the other hand, the sampled signal data are written with the AXI DataMover into the RAM. After the s2mm_finished_intr (interrupt), the Processing System reads the data from the RAM and sends the signal data with UDP/IP to the workstation.

The AID IP core has 2 interrupt output ports, the s2mm_finished_intr and the mm2s_finished_intr. These interrupt signals need a Concat [11] IP core, to route them into the Processing System. The Concat IP core is shown in Figure 125.

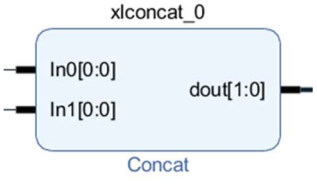


FIGURE 125: CONCAT FOR THE INTERRUPT SIGNALS

To start the AID IP core, the control information must be transmitted from the RAM to the AID IP core. This is done with the AXI DataMover [3]. The PS initializes the transfer and the command data is read from the RAM and transmitted to the AID. The AXI DataMover is also used to write the sampled signal data into the RAM. The AID initializes the transfer and the AXI DataMover writes the signal data into the RAM.

The control information resets or starts the AID IP core. The AID works and transmits the sampled signal data as AXI-Stream to the AXI DataMover. To set up the AID, AXI_GPIO ports are used. Figure 126 shows an AXI_GPIO [6] port. These ports have a memory address, which is shown in the Address Editor of Vivado (Figure 127). The AXI_GPIO ports can be used, to send information from the PS to the user logic. This is used to set the S2MM_MEM_ADDR, S2MM_MEM_ADDR2, MM2S_MEM_ADDR, mm2s_data_length, s2mm_data_length, s2mm_number_pkg and mm2s_startTF. All of these settings can be adjusted in the driver files of the Processing System. The mm2s_startTF signal enables the transfer from the RAM to the AID IP core, which starts the debugging process.

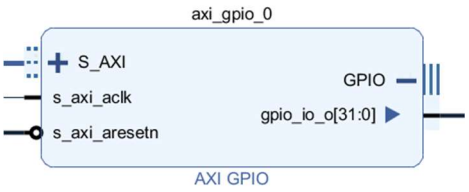


FIGURE 126: AXI_GPIO PIN FOR SETTING UP THE AID IP CORE

Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_1	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_2	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_gpio_3	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_4	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF
axi_gpio_5	S_AXI	Reg	0x4125_0000	64K	0x4125_FFFF
axi_gpio_6	S_AXI	Reg	0x4126_0000	64K	0x4126_FFFF
axi_datamover_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

FIGURE 127: MEMORY ADDRESSES OF THE AXI_GPIO PORTS

To reset all the used IP cores in the design, the Processor System Reset [7] is used 2 times. The first Reset System is used for the 1 MHz reset and the second is used for the 100 MHz reset. The reset signals are active low, which means, when a reset occurs, the signal values gets low and in the normal work condition the signals are high. This is very important to avoid errors and for the design of custom IP cores. The Processor System Reset is shown in Figure 128.

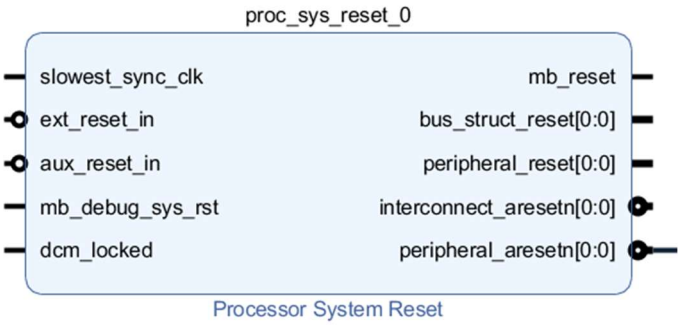


FIGURE 128: PROCESSOR SYSTEM RESET IP CORE

To test the AID, test signals are necessary. The Sig_Gen300 IP core was designed for testing. 300 signal are generated. 20 signals of them are counters, which count with different frequencies. The fastest frequency is 1MHz. Each counter overflows after reaching the highest value and will start at the starting value again. The other 280 signals are constants with the signal number assigned as signal value (e.g. the signal value of signal30 is 30). With this assignment, it is able to test the AID, if the right signals are selected during the signal selection process.

The AID and the Sig_Gen IP core are also available with 40 signals. The design is the same, only these 2 IP cores are different. There is a Xilinx Vivado project (ZB_AID40_PS) with the 40 signals available.

To connect all the IP cores, the Run Connection Automation was used. It automatically generates the AXISmartConnect [12] and the AXIInterconnect [13]. The AXISmartConnect is used to access the RAM. The AXIInterconnect is used to access the AXI_GPIO ports. The finished design is shown in Figure 129. There is also such a design with the 40 signal AID with the 40 signal Sig_Gen IP core and 40 signal AID IP core.

All other IP cores are used from the Xilinx IP catalog.

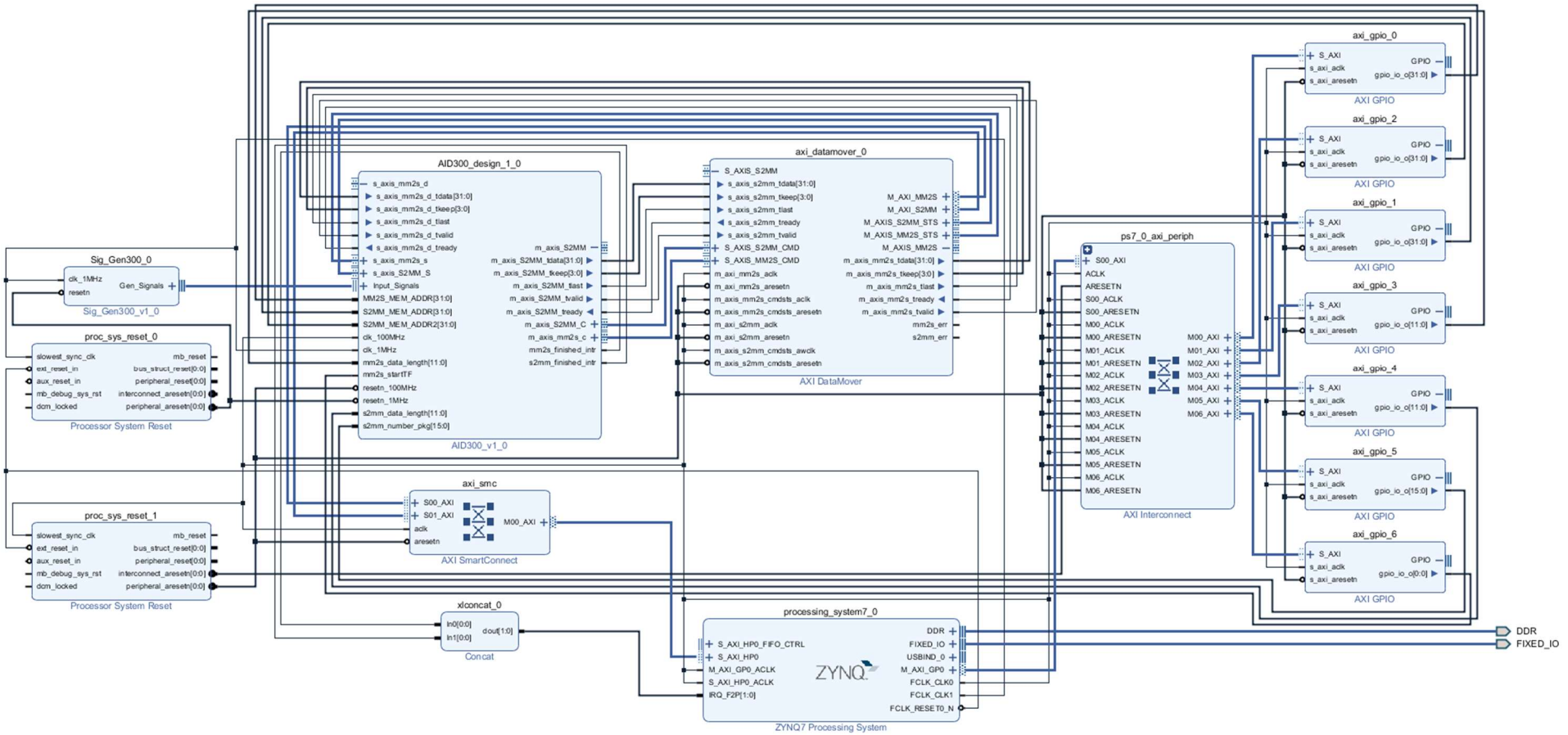


FIGURE 129: ZEDBOARD FPGA DESIGN WITH AID300 AND SIGNAL GENERATOR

After running the design check, the design wrapper was generated. With the wrapper the synthesis, the implementation and the bit stream were generated. A timing constraint file is used for the clock signals. To avoid timing problems, the AID was designed with pipelines to reduce the longest path in the design.

Under the point Export/Export Hardware, the output product was generated with the bit stream.

The SDK automatically creates the hardware platform. The board support package and the application project were created. The application project contains all the written driver files and after building the project, the FPGA was programmed and the design was tested.

With the UART, the C-code was debugged and outputs were created. Figure 130 shows the console outputs, when the application starts. The interrupt system is initialized and the UDP echo server is set up.

```
Serial: (COM4, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)
init interrupt system done.
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
Listening to port: 64000, forwarding to port: 63999
192.168.1.1
Start or Reset
Start or Reset
```

FIGURE 130: START OF THE AID_SW APPLICATION

With the Xilinx ILA core, the internal signals were debugged to verify the correct work. Figure 131 shows the control information to start the debugging process. The AXIS data stream is sent from the RAM to the AID IP core. The transfer starts at 50 us. The signals for the AXIS interface are marked red. After the successful transfer, the interrupt is set by the DataMoveCTL block (orange).

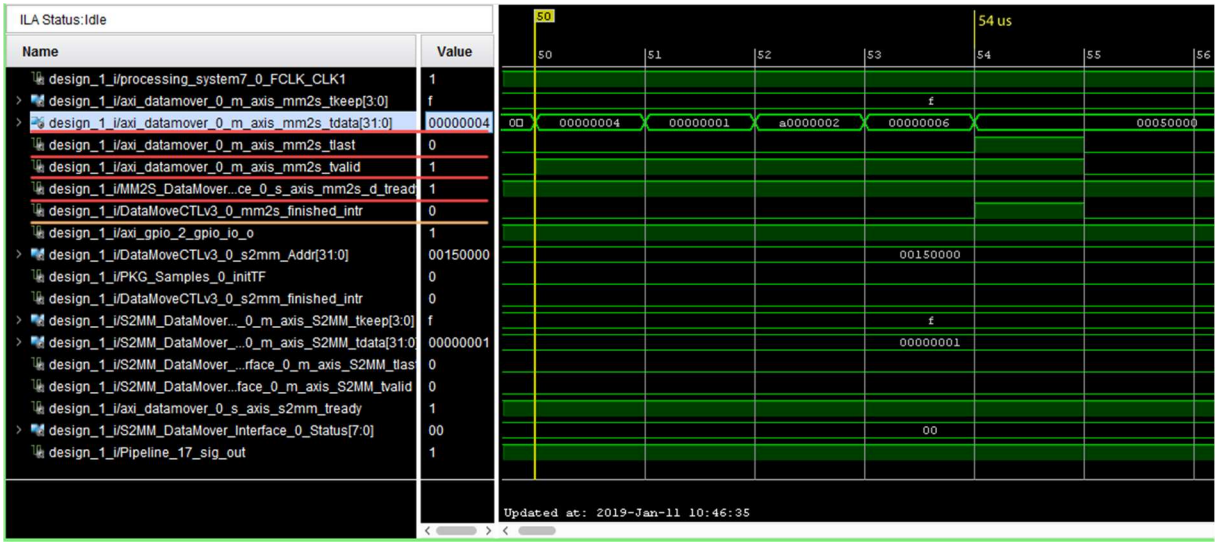


FIGURE 131: AXIS DATA STREAM WITH THE CONTROL INFORMATION TO START THE DEBUGGING PROCESS

Figure 132 shows the AXIS data streams with the signal data, which are written into the RAM. The first transfer is initialized at 3,446 us (red). The signals of the AXIS interface are marked blue. The tready signal is set to 0 after the first data words were transmitted. The tvalid signal stays at 1 until the whole transfer is done. After about 10 us, tready is set to 1 again and the transfer continuous. The tlast signal determines the last data word. After the transfer into the RAM is done, the status is returned (orange). If the transfer was successful, the S2MM memory address is updated (orange). The interrupt is not set (violet), because the number of collected samples for the UDP package is not reached.

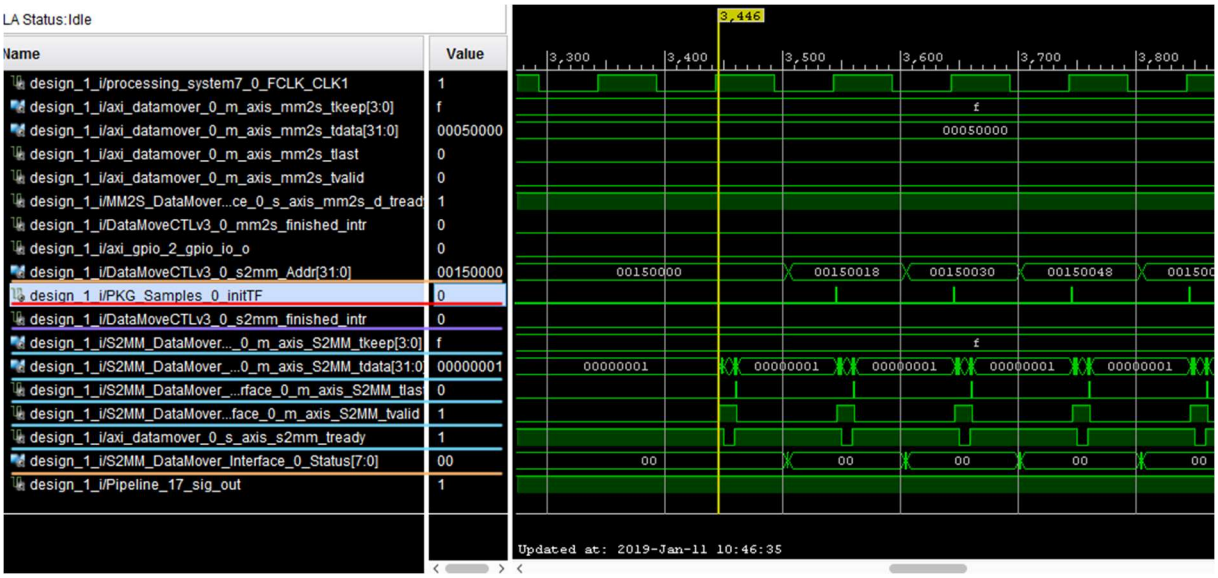


FIGURE 132: AXIS DATA STREAMS WITH THE SIGNAL DATA WRITTEN INTO THE RAM

13 TESTING THE AID IP-CORE ON THE CONTROLLERBOARD

There is already an existing project for the controller board available. The resource files were downloaded from a Git repository. With the TCL [14] script, build.tcl, the whole Vivado project was created.

Due to older IP cores in the TCL script, updates were necessary to create the block design. The AID IP core (AID with 40 signals) and the AXI DataMover were added to the design. 2 ports were added to the interrupt Concat to handle the interrupts. To set up the AID, the AXI_GPIO ports were added and connected.

After a successful design check, the HDL Wrapper was created. After running the synthesis, the implementation and generating the bit stream, the hardware was exported.

With the TCL script, build_SWProj.tcl, the SDK application project with all the resources was created. In the application project a new folder with the driver files for the AID was added. The driver files were modified to include the AID into the existing application project.

The AID is initialized with the following code in the main.c:

```
// global variables definition
u8* DMA_MM2S_Buffer;
u32* DMA_S2MM_Buffer32;
u32* DMA_S2MM_Buffer232;
u8 S2MMaddr1_active;
// add AID in the main function
```

```
#if DC_EN
        // init AID
        print("Init AID UDP connection\n\r");
        AIDConnect_InterruptHandler(&InterruptController, CPU_ID);
        InitAID();
#endif
```

To program the FPGA, a new application project with a Zynq FSBL (First stage boot loader) was added. The bootloader from the fsbl_trenz board was selected and the FSBL application was created. With the FSBL application a new boot image was created to program the Flash. A board restart loaded the boot image. The main application was started with the included AID.

The steps with the FSBL were necessary, because the application project was overwriting the FSBL from the Trenz Board and the application never started.

Figure 133 shows the AID with the AXI_GPIO [6] ports and the AXI DataMover [3]. This IP cores were added, to include the AID into the existing design. The 1 MHz Processor System Reset [7] was already in the design. To set up the AID with the driver files, the AXI_GPIO ports are used. Figure 134 shows the Address Editor with the memory addresses of the design.

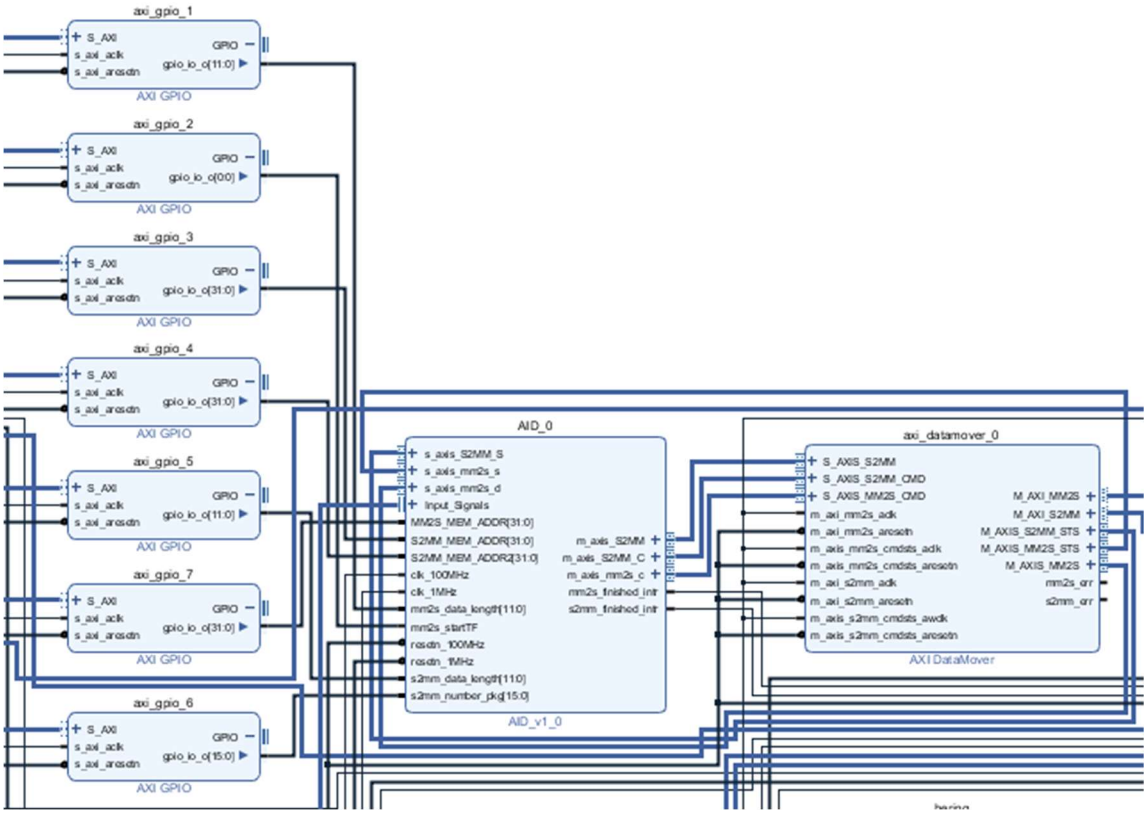


FIGURE 133: CONTROLLER BOARD BLOCK DESIGN WITH THE AID

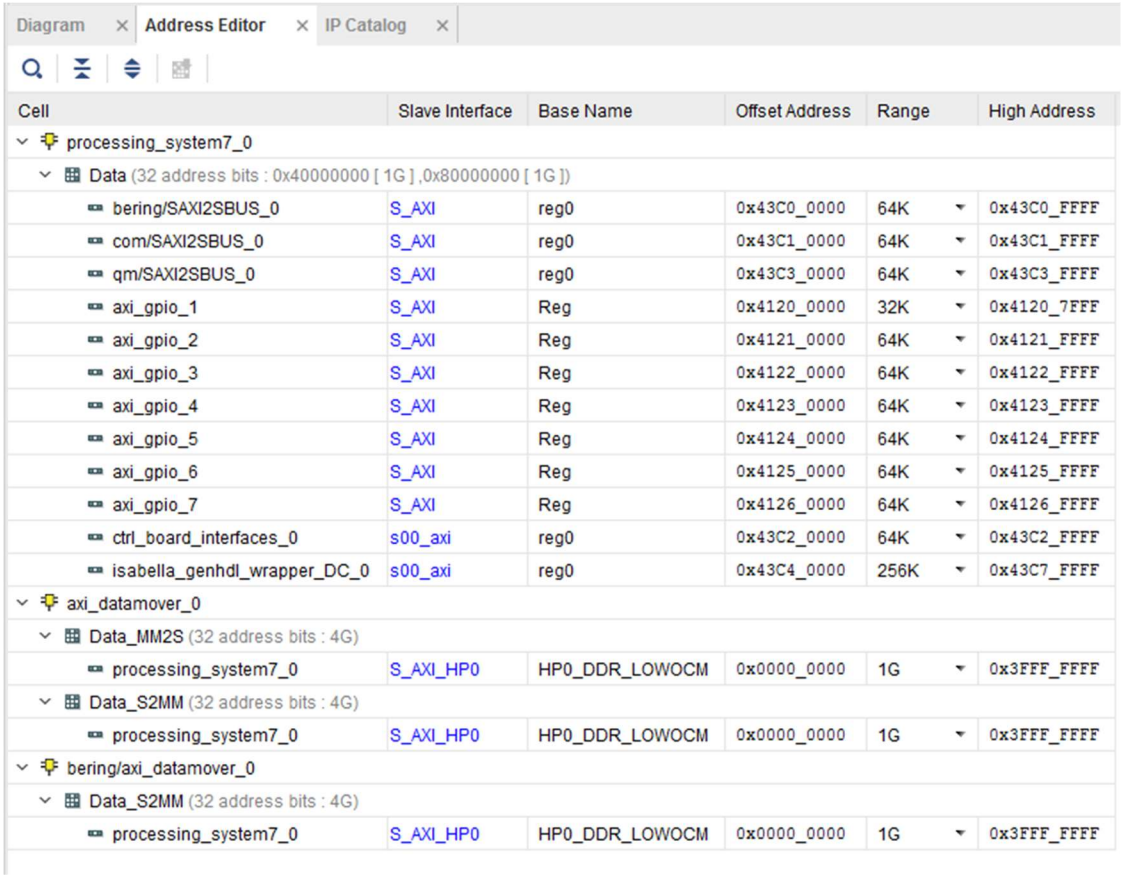


Diagram | Address Editor | IP Catalog

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G] ,0x80000000 [1G])					
bering/SAXI2SBUS_0	S_AXI	reg0	0x43C0_0000	64K	0x43C0_FFFF
com/SAXI2SBUS_0	S_AXI	reg0	0x43C1_0000	64K	0x43C1_FFFF
qm/SAXI2SBUS_0	S_AXI	reg0	0x43C3_0000	64K	0x43C3_FFFF
axi_gpio_1	S_AXI	Reg	0x4120_0000	32K	0x4120_7FFF
axi_gpio_2	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_3	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_gpio_4	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_5	S_AXI	Reg	0x4124_0000	64K	0x4124_FFFF
axi_gpio_6	S_AXI	Reg	0x4125_0000	64K	0x4125_FFFF
axi_gpio_7	S_AXI	Reg	0x4126_0000	64K	0x4126_FFFF
ctrl_board_interfaces_0	s00_axi	reg0	0x43C2_0000	64K	0x43C2_FFFF
isabella_genhdl_wrapper_DC_0	s00_axi	reg0	0x43C4_0000	256K	0x43C7_FFFF
axi_datamover_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
bering/axi_datamover_0					
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF

FIGURE 134: ADDRESS EDITOR OF THE CONTROLLER BOARD FPGA DESIGN

In the existing project, the EtherCat is also used to transmit data. The EtherCat also needs the Processing System to establish a connection. The EtherCat transmission occurs every 100 µs and in the worst case it can last 80 µs. This is a big problem, because 80% of the processor is used for EtherCat.

Due to the processor utilization of the EtherCat, interrupts from the AID are missed. This is a problem because UDP packages are not sent and the samples are lost. This causes gap in the charts.

During the tests with a sample rate of 1 MHz and 500 kHz, some UDP packages were not sent, due to EtherCat. EtherCat was still in idle mode but when it is running and transmitting data, the performance will be much worse.

Due to the missing AXI-Ethernet IP core, which provides an Ethernet stack, the Processing System was used to establish the UDP connection. This is not the best option to establish a fast UDP connection, because the Processing System is slow. The AXI-Ethernet [4] IP core would be the best solution to solve this problem. The AXI-Ethernet IP uses the TEMAC [5] core, which requires a license to add it to the design. Maybe this will be done in the future.

Figure 135 shows the debugging test with a sample rate of 250 kHz and 31744 samples. With the zoom, the generated sinus signals are seen. The sinus signals are generated from the black box, where usually the inverter software is included. If no inverter software is included, the block box generates sinus signals. They are used to debug existing IP cores and to test different I/O operations and communications with other devices.

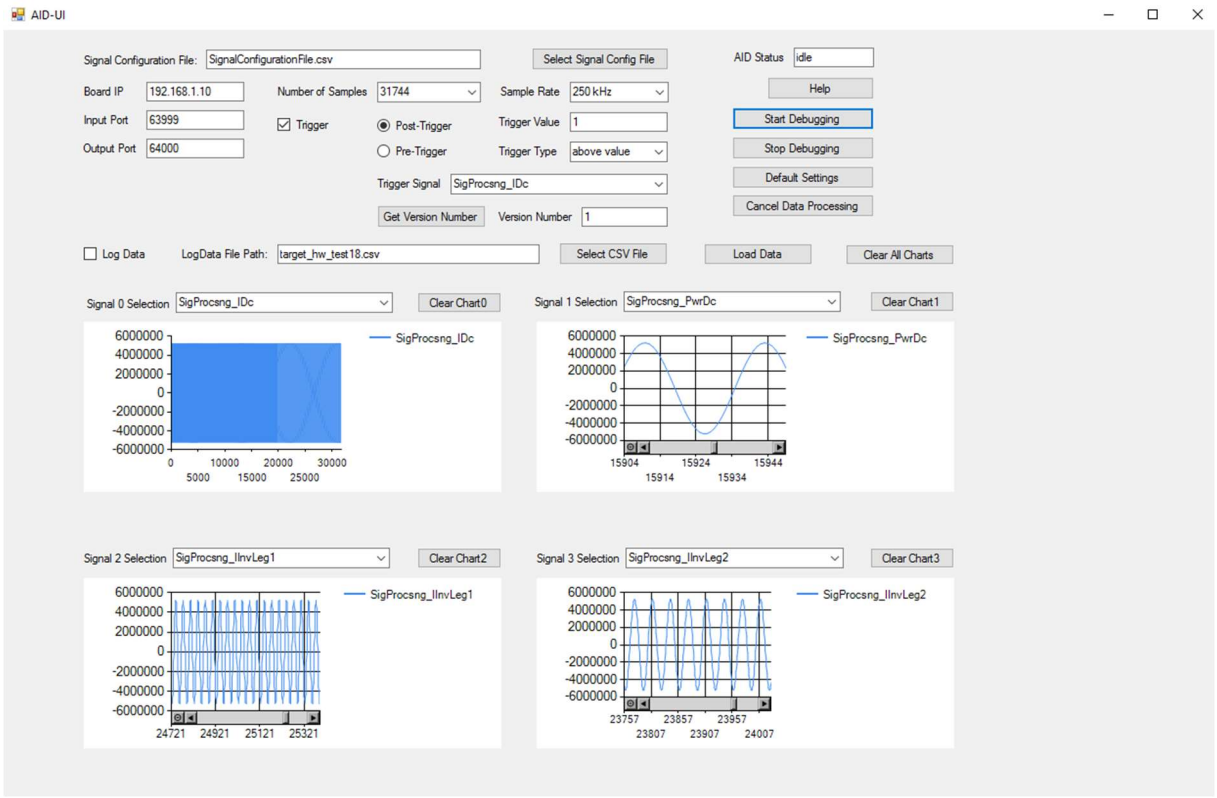


FIGURE 135: DEBUGGING TEST WITH A SAMPLE RATE OF 250KHZ

Figure 136 shows the debugging test with a sample rate of 1.6 kHz and infinity mode. In the Explorer, the generated log files are shown. The maximum sample number per file was set to 249856. Therefore, 2 log files were created. Currently the number of samples per file is set to 999424.

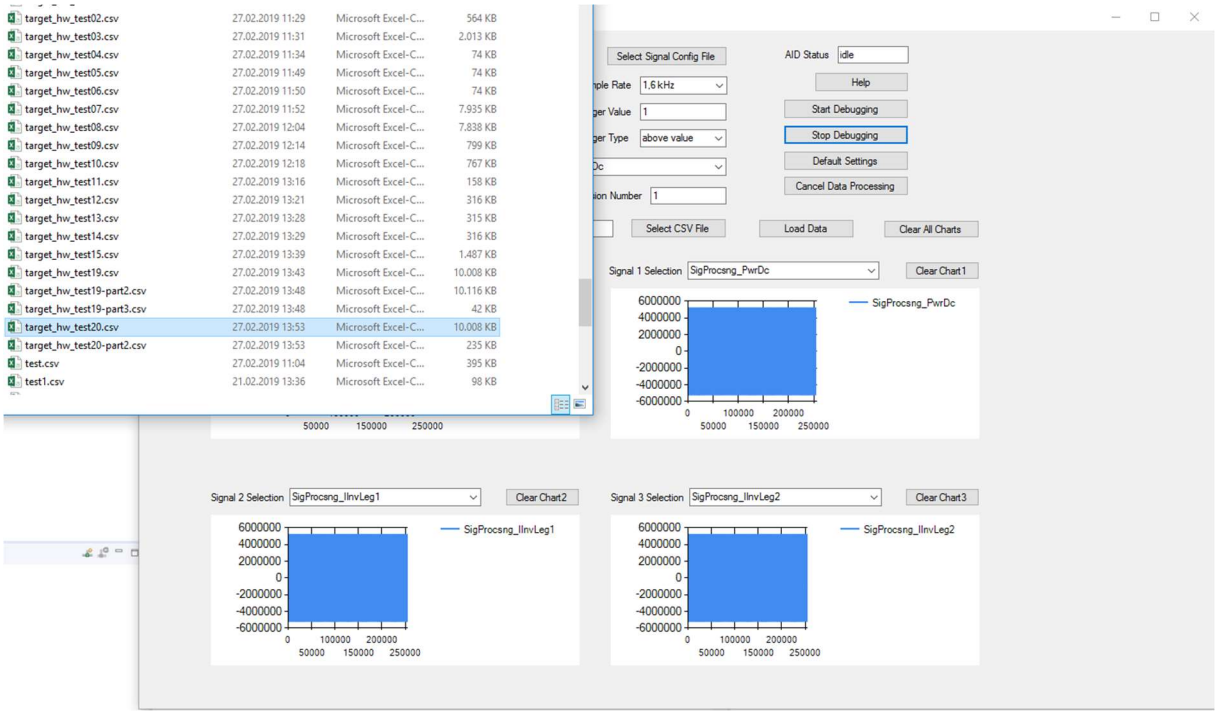


FIGURE 136: DEBUGGING TEST WITH A SAMPLE RATE OF 1.6 KHZ AND INFINITY MODE

Due to UDP, the packages may arrive in the wrong order. To increase the performance of the user-interface, this is not handled. Figure 137 shows the charts, when packages are received in the wrong order.

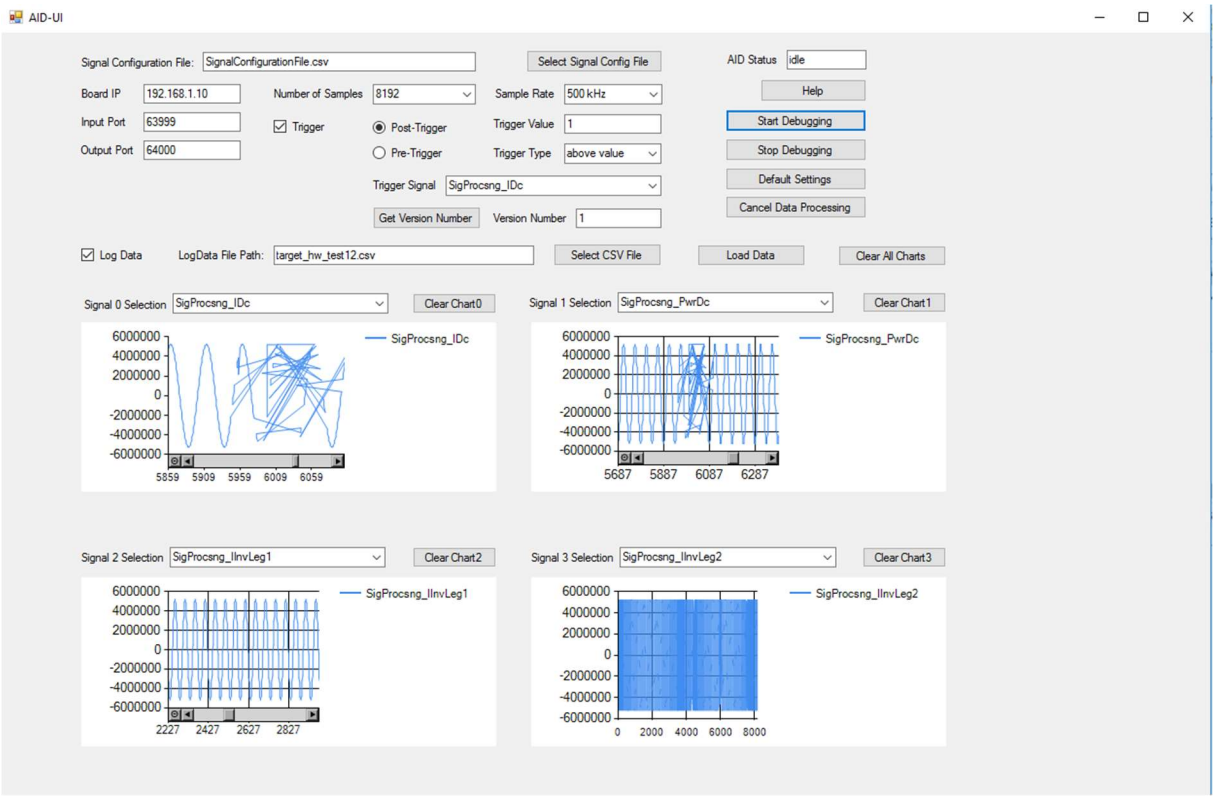


FIGURE 137: WRONG ORDER OF THE UDP PACKAGES

Figure 138 and Figure 139 show missing UDP packages in Wireshark. Due to the high processor utilization with EtherCat, interrupts of the AID are missed and the UDP packages are not sent. The sample frequency was 500 kHz and 64 samples (2 UDP packages) are missing between the packages 455 and 456. The sample number in the UDP package 455 starts with 0x00001D41 (7489) and the sample number in the UDP package 456 starts with 0x00001DA1 (7585). This behavior appeared through the whole debugging process. At 1 MHz sample rate this behavior is even worse. There are 4 or more UDP packages not sent, which are at least 128 missing samples.

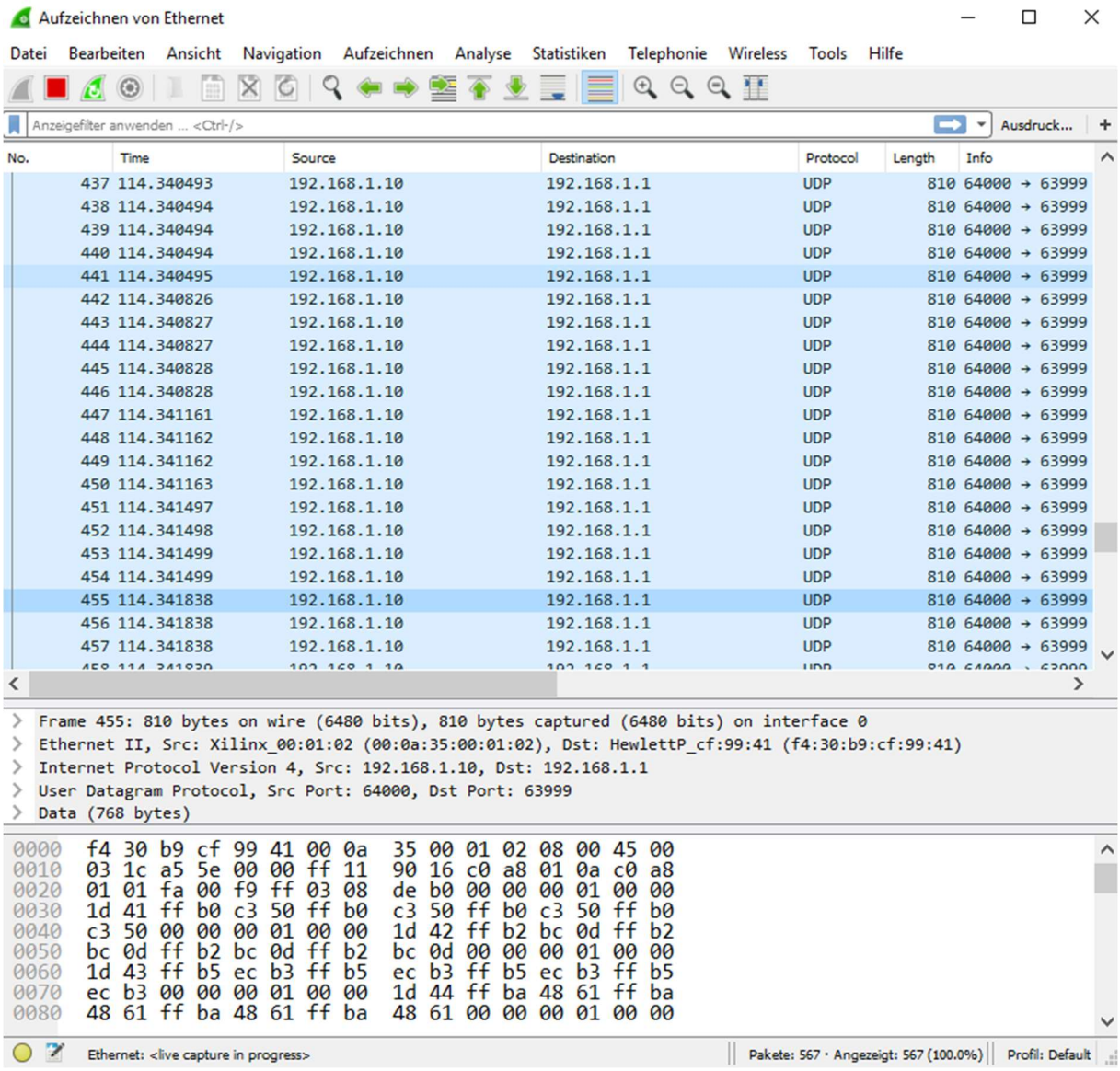


FIGURE 138: UDP PACKAGES MISSING IN WIRESHARK

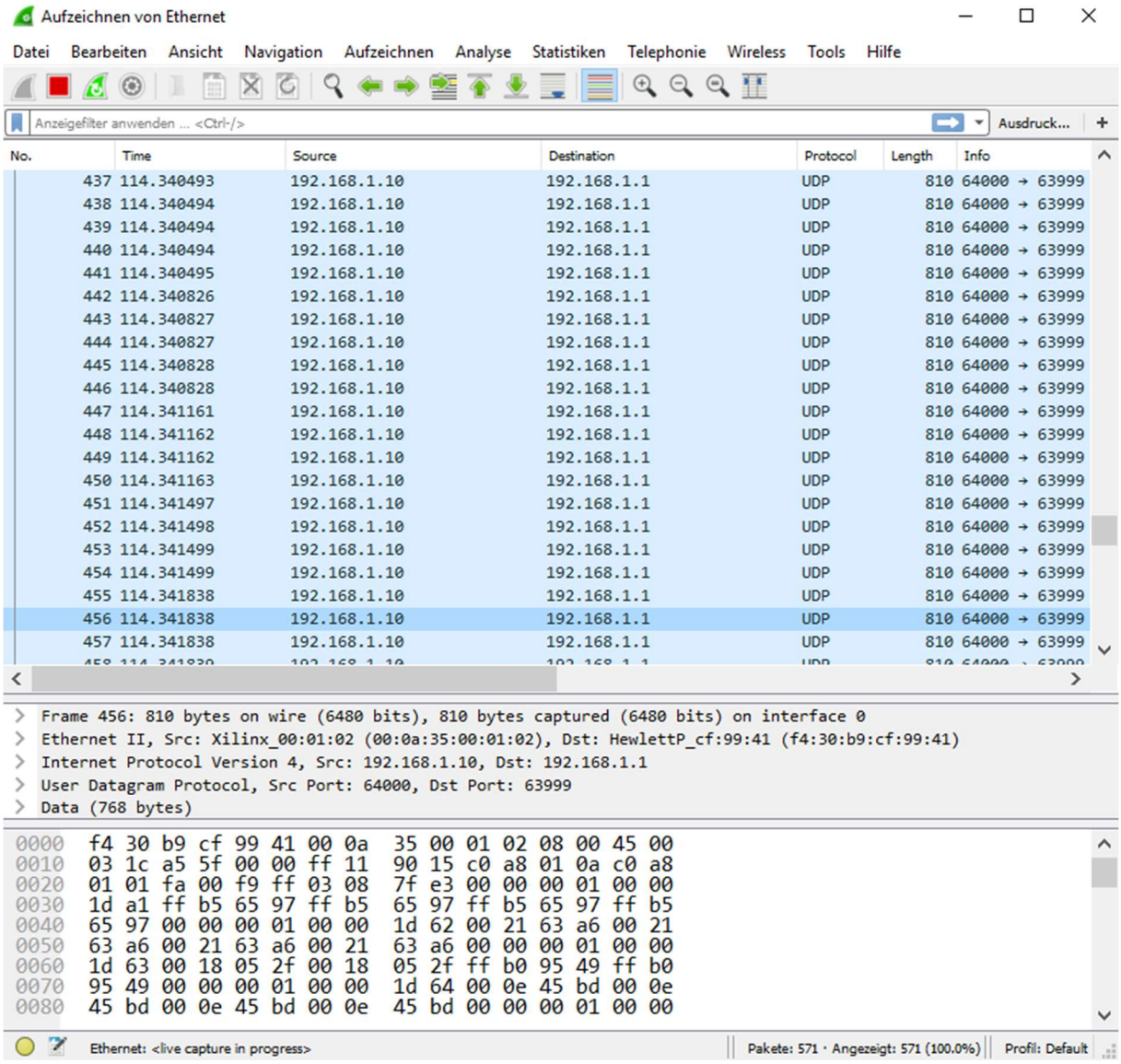


FIGURE 139: UDP PACKAGES MISSING IN WIRESHARK

Figure 140 shows the debugging process with a sample rate of 500 kHz and with 20480 samples. There are small gaps in the charts. The biggest gap is about 1000 samples. At a frequency of 500 kHz not all packages are sent. This generates additionally gaps in the charts. The receiving thread has also a problem to process the incoming UDP packages, which creates the bigger gap.

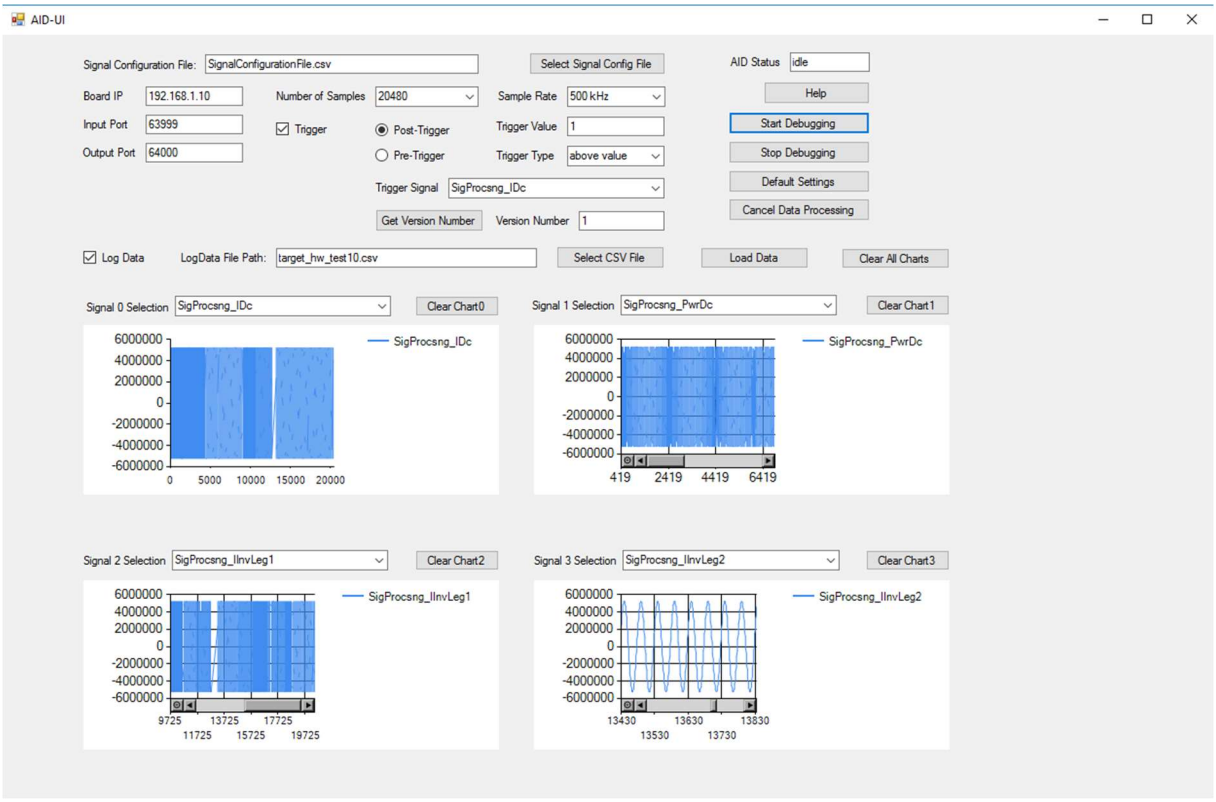


FIGURE 140: DEBUGGING PROCESS WITH GAPS AT A SAMPLE RATE OF 500 KHZ

The debugging process with a higher number of samples, creates bigger gaps at the same sample rate. This is shown in Figure 141. The sample rate is still 500 kHz and the number of samples is 31744.

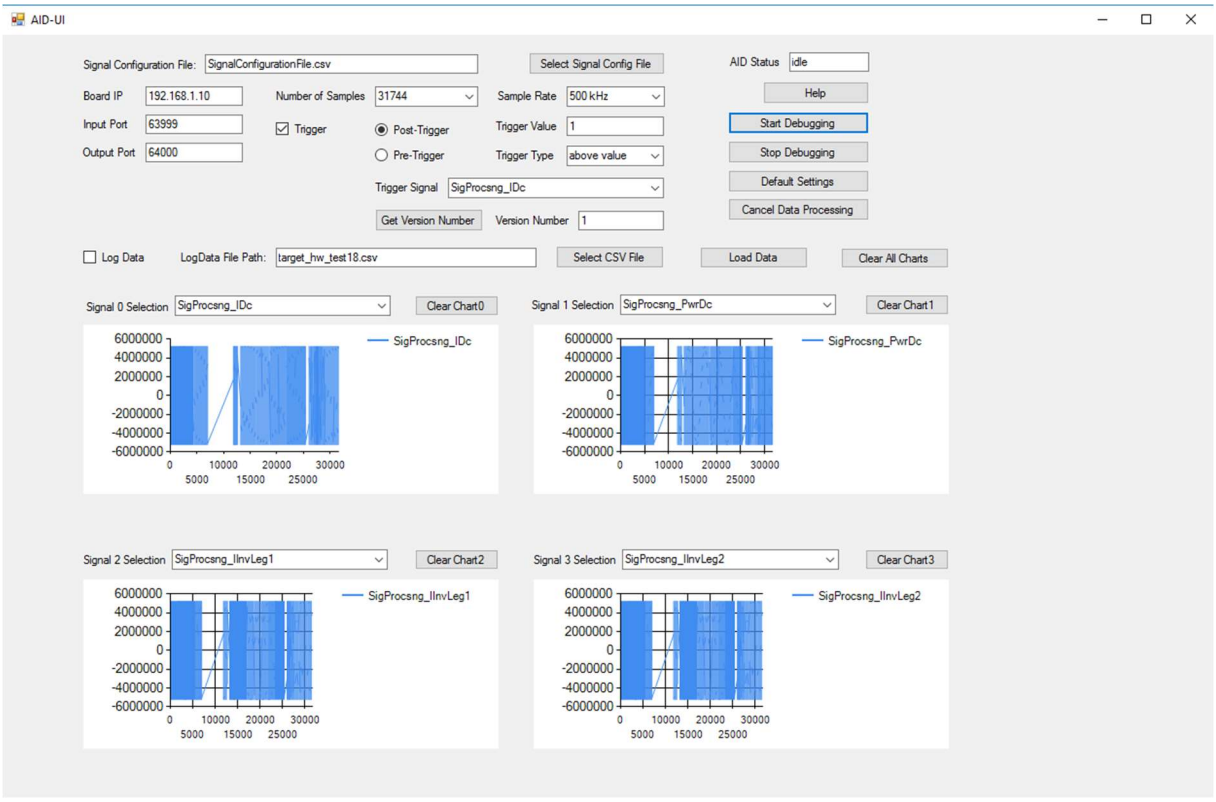


FIGURE 141: DEBUGGING PROCESS WITH GAPS AT A SAMPLE RATE OF 500 KHZ

These gaps can also appear with a higher number of samples at a sample rate of 250 kHz and 200 kHz. Mostly there are no gaps in the charts. The gaps may be caused by the DMA of the workstation as well, when the data is swapped from the RAM to the HDD or SSD. The receiving thread needs more time to fill the FIFO queue with the received data, when the data is written to the HDD (in the virtual memory).

14 USED TOOLS

Matlab-Simulink

The DebugCoreModule was created with MATLAB[39] Simulink[40] 2017b. With MATLAB, it is possible to convert the MATLAB Simulink model to HDL-Code. It is also possible to simulate the Debug-Core.

MATLAB -Toolboxes:

- MATLAB
- Simulink
- Simulink Coder
- MATLAB Coder
- Embedded Coder
- HDL Coder
- HDL Verifier

Xilinx-Vivado

The AID IP core and the FPGA designs were created with Xilinx Vivado[41] 2017.2.

Xilinx Vivado is a tool for FPGA development and after generating the HDL-Code with MATLAB, the HDL-Code can be synthesized with Vivado to generate the bit stream for the hardware.

Vivado is a design and development tool from Xilinx. This tool supports high level design, verification and implementation for FPGA, DSP and SoC designs. With the high-level synthesis, IP generator, logic simulation, mixed language simulator, verification IP and programming and debug environment, Vivado has a lot of benefits and is wide spread for this kind of hardware development.

[39] MATALB, https://de.mathworks.com/products/matlab.html
[40] Simulink, https://de.mathworks.com/products/simulink.html
[41] Xilinx Vivado, https://www.xilinx.com/products/design-tools/vivado.html

LabVIEW

The LabVIEW user-interface was created with NI LabVIEW 2017.

NI LabVIEW[42] provides:

- A fast recording of measurement data through integration of hardware.
- A graphical system abstraction, which offers access to collected data for validation of hardware connections.
- A graphical programming system for the automation of systems and for repeatable measurements.

LabVIEW was used to program the first user-interface.

Visual Studio

The C# user-interface was created with Visual Studio 2017. Microsoft Visual Studio[43] is a development tool for software development. It provides the tools for software development with different programming languages like C#, Python, C++, C…

C# was used to program the second user-interface, to gain more performance.

Wireshark

Wireshark 2.6.5 was used to analyze the UDP packages. Wireshark[44] is a tool to analyze the network connections. It monitors incoming and outgoing packages. This tool was needed to verify the sent and received UDP packages from the communication between the user-interface and the AID.

Visio

Visio 2016 was used to create block structures of the design and technical images. Microsoft Visio[45] is a visualization tool from Microsoft. It is possible to create flow diagrams, business processes and technical images. Visio was used to create technical images.

[42] LabVIEW, http://ni.com/de-at/shop/labview/labview-details.html
[43] Microsoft Visual Studio, https://visualstudio.microsoft.com
[44] Wireshark, https://www.wireshark.org
[45] Microsoft Visio, https://products.office.com/de-at/visio/flowchart-software

15 USED TEST-HARDWARE

ZedBoard Development Kit

- ZedBoard[46]
- 12V Power supply
- Micro USB cable
- USB Adapter: Male Micro-B to Female Standard-A
- 4 GB SD Card
- Xilinx Vivado® Design Edition license voucher (device locked to 7Z020)
- Getting started guide
- Downloadable documentation and reference designs
- MathWorks Getting Started Package (optional)

The ZedBoard is the hardware, where the Debug-Core will run. The Debug-Core was also tested on the ZedBoard. The Development Kit also includes a Xilinx Vivado license (locked to 7Z020) for the Synthesis. To use the ZedBoard, it is necessary to install the Embedded Coder® Support Package for Xilinx®Zynq®-7000 Platform in MATLAB. With this package, it is possible to use blocks, which are provided by the hardware.

Controller Board

The controller board was developed by AVL List GmbH and FH Kapfenberg. It contains a Zynq-7000 Kintex-7 FPGA, a Trenz Board TE0782-02 and several peripherals for different applications. The different inverters and controllers are running on the FPGA and the AID will be included into the design to debug the internal signals.

- Zynq-7000 Kintex-7 FPGA (Xc7z100ffg900)
- Trenz Board TE0782-02

Trenz Board TE0782-02

The Trenz Board TE0782-02[47] is a High-Performance Xilinx Zynq Z-7100 Module.

Properties:

- Xilinx Zynq-7100 SoC XC7Z100-2FFG900I
- Dual ARM Cortex-A9 MPCore
- Real-Time
- 2x Hi-Speed USB 2.0 ULPI Transceiver PHY
- 2x Gigabit Ethernet Transceiver PHY
- 2x Ethernet MAC Address EEPROM
- 1 GB DDR3 SDRAM
- 32 MB QSPI Flash-Memory

---

[46] ZedBoard, http://zedboard.org/product/zedboard
[47] Trenz Board, https://shop.trenz-electronic.de/de/TE0782-02-100-2I-High-Performance-Xilinx-Zynq-Z-7100-Modul-industriell-8-5-x-8-5-cm

- 4 GB eMMC (optional up to 64 GB)
- Optional 2x 8MB HyperRAM (maximum 2 x 32 MB HyperRAM)
- Si5338 PLL for GTX clocking
- Plug-on-Module with 3 x 160-Pin High-Speed strips
- 16 GTX high-performance transceiver lanes
- 254 FPGA I/O (125 LVDS pairs)
- On-board high power DC-DC-Converter
- System management and power sequencing
- eFUSE bit-stream encryption
- AES bit-stream encryption
- Distributed power pins
- Temperature -40°C to +85°C

Notebook HP EliteBook 840

- 512 GB M.2 SSD
- 8GB (1x8GB) 2133MHz DDR4 RAM
- 6th Gen Intel® Core™ i5-6300U Processor (2.4 GHz, 3MB, Dual Core)
- Intel Integrated HD 520 Graphics

This notebook belongs to AVL List GmbH and was used to develop the IP cores (with Xilinx Vivado and MATLAB), the LabVIEW user-interface. It was also used to test the AID with the target hardware and both user-interfaces.

Notebook MSI GE73VR 7RF Raider

- 256 GB SSD
- 16GB 2400MHz DDR4 RAM
- 7th Gen Intel® Core™ i7-7700HQ Processor (2.8 GHz, 6MB, Quad Core)
- GeForce® GTX 1070

This notebook was used to make tests with the C# user-interface.

16 CONCLUSION

For the development of FPGA based products, debugging of internal signals is necessary to detect errors or to visualize signals of the FPGA design. Xilinx Vivado already offers an Integrated Logic Analyzer (ILA) to debug signals of the design. Sometimes, custom Debug-Cores are necessary for prototype development, like the AID – Advanced Inverter Debugger. The AID has 300 possible signal inputs. It can dynamically select 4 signals out of them for the debugging process. The debugging process is controlled by a user-interface at a workstation. The adjusted debugging parameters are sent from the workstation to the Debug-Core on the FPGA. The communication is done with UDP/IP. The Debug-Core starts the debugging process with the adjusted parameters. The sampled signal data is sent from the FPGA to the workstation and monitored with the user-interface. Optional, the signal data can be logged in csv files.

To lower the resource usage on the FPGA, the AID is also available with 40 possible signal inputs. 40 signals are enough to debug the different inverters and the advantage of dynamically selecting the signals for the debugging process is still present.

For the development of the AID, different concepts were considered and compared. In most cases, data is stored on the FPGA and big data packages are sent to the workstation. In the case of this master thesis, the data is transmitted with UDP/IP. UDP/IP was used for small data packages.

The first approach was to stream the sampled signal data with the sample frequency to the workstation. Therefore, the direct connection from the Ethernet adapter to the AXI-Stream interface with the AXI-Ethernet IP core should be used. This IP core is provided by Xilinx and requires a license for the included TEMAC (Tri-Mode Ethernet MAC). However, this license was not available and therefore the Processing System had to be used to establish the UDP/IP connection between the FPGA and the workstation.

At the maximum sample frequency of 1 MHz, the Processing System was not able to stream the sampled signal data with the sample frequency anymore. The Processing System is too slow to process each interrupt from the AID. Therefore, samples are collected to build bigger UDP packages. To collect the samples, 2 memory addresses are alternately used to avoid simultaneous memory access.

Despite the sample collection, the UDP packages are sent too fast for the receiver workstation. The LabVIEW user-interface works for slower sample frequencies but at higher sample frequencies the data processing is not fast enough and samples get lost. Furthermore, the file handling during runtime is very complex. It was not possible to dynamically create several log files when the sample number is very high (millions). To increase the performance of the data processing, wait functions were added to reduce the number of executions of while-loops in LabVIEW, which are not used for data processing. This increased the performance, however data loss still occurs.

The C# user-interface was developed to dynamically create log files and to increase the performance of the data processing. To process the incoming UDP packages, 2 threads are used. The receiving thread handles the incoming UDP packages and writes the data into a FIFO queue. The processing thread reads the data from the FIFO queue and updates the charts and writes the data into the log files. The performance of the C# user-interface is much better than the LabVIEW user-interface, however sample loss also occurs at high sample frequencies. The charts of the C# library update very slowly and are not suited for that kind of application. Licensed libraries for fast data monitoring would be necessary to increase the performance.

During the tests with the Controller Board, the Processing System was mostly used for EtherCat and other communications. EtherCat has a very high processor utilization in idle mode. When EtherCat is transmitting data, the processor utilization gets even worse. Due to the high utilization, the Processing System has not enough time to handle the AID interrupts. If interrupts are missed, UDP packages are not sent anymore and samples get lost.

The approach to stream the data with UDP/IP works with a lower number of samples and lower sample frequencies. It is possible to debug signals with a sample frequency of 1 MHz with a lower number of samples. The communication with the Processing System is a big disadvantage because the Processing System also handles other communications. In the future, the AXI-Ethernet IP core can be used for the UDP/IP connection. With this IP core, the communication can be done without the Processing System. At the workstation, fast data processing is necessary to handle the small data transfers with UDP/IP.

The performance of the user-interfaces might be better, if the received UDP packages are saved directly into files. When the debugging process is done, the signal data is processed and monitored afterwards. This solution would not require fast data processing libraries. This is also a task for the future.

The performance of the LabVIEW user-interface can be increased by using a real-time operating system and the LabVIEW extension for real-time applications. The data processing can be done with event triggered while loops, which are faster than the functions from the standard LabVIEW version. A big disadvantage of this solutions would be, that a real-time operating system is necessary and no normal workstation can be used, to process the received UDP packages.

17 REFERENCES

[1] Xilinx Inc, "Processing System 7 v5.5," 10 May 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_5/pg082-processing-system7.pdf. [Accessed 06 Mai 2018].

[2] Xilinx Inc, "AXI Reference Guide," Xilinx, 15 07 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf. [Accessed 13 December 2017].

[3] Xilinx Inc, "AXI DataMover v5.1," 5 April 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v5_1/pg022_axi_datamover.pdf. [Accessed 27 June 2018].

[4] Xilinx Inc, "AXI 1G/2.5G Ethernet Subsystem v7.0," 5 April 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_0/pg138-axi-ethernet.pdf. [Accessed 9 October 2018].

[5] Xilinx Inc, "Tri-Mode Ethernet MAC v9.0," 4 April 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/tri_mode_ethernet_mac/v9_0/pg051-tri-mode-eth-mac.pdf. [Accessed 15 October 2018].

[6] Xilinx Inc, "AXI GPIO v2.0," 5 October 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf. [Accessed 15 July 2018].

[7] Xilinx Inc, "Processor System Reset Module v5.0," 18 November 2015. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf. [Accessed 13 Mai 2018].

[8] Xilinx Inc, "Using Constraints," 4 September 2012. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug903-vivado-using-constraints.pdf. [Accessed 1 August 2018].

[9] Microsoft, "Excel specifications and limits," [Online]. Available: https://support.office.com/en-us/article/excel-specifications-and-limits-1672b34d-7043-467e-8e27-269d656771c3. [Accessed 3 November 2018].

[10] Wikipedia, "UDP port numbers," [Online]. Available: https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers. [Accessed 12 March 2018].

[11] Xilinx Inc, "LogiCORE IP Concat v2.1," 6 April 2016. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xilinx_com_ip_xlconcat/v2_1/pb041-xilinx-com-ip-xlconcat.pdf. [Accessed 7 Mai 2018].

[12] Xilinx Inc, "SmartConnect v1.0," 4 March 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf. [Accessed 5 March 2019].

[13] Xilinx Inc, "AXI Interconnect v2.1," 20 December 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf. [Accessed 23 September 2018].

[14] Xilinx Inc, "Using Tcl Scripting," 5 April 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug894-vivado-tcl-scripting.pdf. [Accessed 2 January 2019].

[15] Xilinx Inc, "System-Level Design Entry," 4 May 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug895-vivado-system-level-design-entry.pdf. [Accessed 2 January 2019].

[16] Xilinx Inc, "Xilinx Vivado," [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[17] Xilinx Inc, "Xilinx SDK Doc Index," [Online]. Available: https://www.xilinx.com/html_docs/xilinx2016_2/SDK_Doc/index.html.

[18] AVNET, "ZedBoard," [Online]. Available: http://zedboard.org/product/zedboard. [Accessed 13 12 2017].

[19] Wireshark-Community, "Wireshark," [Online]. Available: https://www.wireshark.org/. [Accessed 12 August 2018].

[20] National Instruments, "LabVIEW," [Online]. Available: http://www.ni.com/de-at/shop/labview/labview-details.html. [Accessed 13 12 2017].

[21] FH Kapfenberg, "FH Kapfenberg," [Online]. Available: https://www.uni.at/uni/fh-joanneum-kapfenberg/.

[22] MathWorks, "MATLAB," [Online]. Available: https://de.mathworks.com/products/matlab.html.

[23] MathWorks, "Simulink," [Online]. Available: https://de.mathworks.com/products/simulink.html.

[24] AVL List GmbH, "AVL List GmbH," [Online]. Available: https://www.avl.com/.

[25] Trenz Electronic, "Trenz Board," [Online]. Available: https://shop.trenz-electronic.de/de/TE0782-02-100-2I-High-Performance-Xilinx-Zynq-Z-7100-Modul-industriell-8-5-x-8-5-cm?c=360.

[26] National Instruments, "LabVIEW real-time," [Online]. Available: https://www.ni.com/de-at/shop/data-acquisition-and-control/add-ons-for-data-acquisition-and-control/what-is-labview-real-time-module.html.

[27] Microsoft, "Visual Studio," [Online]. Available: https://visualstudio.microsoft.com.

[28] Microsoft, "Visio," [Online]. Available: https://products.office.com/de-at/visio/flowchart-software.

18 LIST OF FIGURES

19 LIST OF TABLES

20 APPENDIX

TABLE 6: MM2S_DATAMOVER_INTERFACE INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
s_axis_mm2s_d		Slave interface for the AXIS data stream from the AXI DataMover
s_axis_mm2s_s		Slave interface for the AXIS status stream of the AXI DataMover
axis_aclk	1	Clock for the AXIS interfaces
axis_resetn	1	Resetn for the AXIS interfaces
BTT	23	Bytes to transfer
Addr	32	Memory start address
initTf	1	Initializes the transfer of the command data

TABLE 7: MM2S_DATAMOVER_INTERFACE OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
m_axis_mm2s_c		Master interface for the AXIS command data for the AXI DataMover
m_axis_umm2s_d		Master interface for the AXIS user data for the PL
Status	8	Status of the AXI DataMover transfer

TABLE 8: S2MM_DATAMOVER_INTERFACE INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
s_axis_S2MM_S		Slave interface for the AXIS status stream of the AXI DataMover
s_axis_uS2MM		Slave interface for the AXIS user data
axis_aclk	1	Clock for the AXIS interfaces
axis_resetn	1	Resetn for the AXIS interfaces
BTT	23	Bytes to transfer
Addr	32	Memory start address
initTf	1	Initializes the transfer of the command data

TABLE 9: S2MM_DATAMOVER_INTERFACE OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
m_axis_S2MM_C		Master interface for the AXIS command data for the AXI DataMover
m_axis_S2MM		Master interface for the AXIS data stream to the AXI DataMover
Status	8	Status of AXI DataMover transfer

TABLE 10: DATAMOVECTL INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
s2mm_data_length	12	Length of data stream, which will be written into the RAM
s2mm_Status	8	Status of the transfer into the RAM
s2mm_startTF	1	Initialization of the S2MM transfer
S2MM_MEM_ADDR	32	First S2MM start memory address. Only active if C_PS_CONTROL is enabled, otherwise the memory address of the IP Settings is used
S2MM_MEM_ADDR2	32	Second S2MM start memory address. Only active if C_PS_CONTROL is enabled, otherwise the memory address of the IP Setting is used.
s2mm_number_pkg	16	Number of samples, which will be stored into the RAM till the s2mm_finished_intr is set. Only active if C_PS_CONTROL is enabled, otherwise C_NUMBER_PKG of the IP Settings is used.
mm2s_Status	8	Status of the transfer from the RAM
mm2s_startTF	1	Initialization of the MM2S transfer
mm2s_data_length	12	Length of the data stream, which will be read from the RAM
MM2S_MEM_ADDR	32	MM2S start memory address. Only active if C_PS_CONTROL is enabled, otherwise the memory address of the IP Settings is used.
aclk	1	100 MHz clock
aresetn	1	Resetn, active low
Resetn_DC	1	Resetn from the Debug-Core module, when the debugging process is finished, the DataMoveCTL block is reset. Active low

TABLE 11: DATAMOVECTL OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
s2mm_BTT	23	S2MM Bytes to Transfer
s2mm_Addr	32	S2MM start memory address
s2mm_initTF	1	Initialization of the S2MM transfer
s2mm_finished_intr	1	Finish interrupt. It will be set when the transfer into the RAM was successful and the adjusted number of samples was reached.
mm2s_BTT	32	MM2S Bytes to Transfer
mm2s_Addr	8	MM2S start memory address
mm2s_initTF	8	Initialization of the MM2S transfer
mm2s_finished_intr	1	Finish interrupt. It will be set when the transfer from the RAM was successful.

TABLE 12: DATAMOVECTL IP SETTINGS

Setting	Bits	Description
C_ADDR_WIDTH	32	Address width of the memory addresses (set to 32)
C_BTT	23	Bytes to transfer width (set to 23)
C_MM2S_MEM_ADDR	32	MM2S memory start address (default value is 0x00120000)
C_NUMBER_PKG	16	Number of Samples to collect for the UDP package (value 1-32768), currently set to 32
C_S2MM_MEM_ADDR	32	First S2MM memory start address (default value is 0x00140000)
C_S2MM_MEM_ADDR2	32	Second S2MM memory start address (default value is 0x00160000)
C_PS_CONTROL	1	Enable Processing System control (checked is PS control enabled)

TABLE 13: UNPKGMODULE INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
S_AXI4S		Slave interface for the AXIS data stream
clk_1MHz	1	1 MHz clock
aresetn_1MHz	1	Resetn to corresponding 1 MHz clock
aclk_100MHz	1	100 MHz clock for the AXIS interface
aresetn	1	Resetn to corresponding 100 MHz clock
Com_PS_en	1	Enables the Communication with Processing System, currently in use

TABLE 14: UNPKGMODULE OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
pkg_ctl	16	Package type (not used)
NumOfSamples	32	Number of Samples to be debugged
SampleRate	16	Sample Rate for the debugging process
CMD	8	Command signal
TriggerTyp	8	Trigger Type for the debugging process
TriggerValue	16	Trigger Value for the debugging process
Signal0	16	Chosen signal to debug and also trigger signal
Signal1	16	Chosen signal 2 to debug
Signal2	16	Chosen signal 3 to debug
Signal3	16	Chosen signal 4 to debug
start_1MHz	1	For debugging

TABLE 15: UNPKG_UDP_CTL_UNIT INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
S_AXI4S_tlast	1	AXIS TLAST of the S_AXI4S interface. This signal is used to signal the end of the data transfer.
S_AXI4S_tvalid	1	AXIS TVALID of the S_AXI4S interface. This signal is used to get the valid data.
aclk	1	100 MHz clock
aresetn	1	Resetn to the corresponding 100 MHz clock
com_PS_en	1	Enables the module for the communication with the integrated Ethernet interface of the ARM-Processor. Is currently set to 1.

TABLE 16: UNCP_UDP_CTL_UNIT OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
S_AXI4S_tready	1	AXIS TREADY of the S_AXI4S interface is always 1 (always ready to receive data)
en_pkg_ctl	1	Enable signal for package type
en_NumOfSamples	1	Enable signal for number of samples
en_SR_CMD_TT	1	Enable signal for sample rate, command and trigger type
en_TV_Sig1	1	Enable signal for trigger value and first signal
en_Sig2_Sig3	1	Enable signal for second and third signal
en_Sig4	1	Enable signal for fourth chosen signal

TABLE 17: UNPKG_UDP_DATA INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
S_AXI4S_tdata	32	AXIS tdata of the S_AXI4S interface
aclk	1	100 MHz clock
aresetn	1	Resetn to the corresponding 100 MHz clock
com_PS_en	1	Enables the module for the communication with the ARM-Processor and the integrated Ethernet interface. Is currently set to 1.
en_pkg_ctl	1	Not used
en_NumOfSamples	1	Enable signal for number of samples
en_SR_CMD_TT	1	Enable signal for sample rate, command and trigger type
en_TV_Sig1	1	Enable signal for trigger value and first chosen signal
en_Sig2_Sig3	1	Enable signal for second and third chosen signal
en_Sig4	1	Enable signal for fourth chosen signal

TABLE 18: UNPKG_UDP_DATA OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
CMD	8	Command Byte, gives information to start debugging, stop debugging and if pre- or post-trigger is active
NumOfSamples	32	Number of Samples
SampleRate	16	Sample Rate
Signal1	16	First chosen signal is also trigger signal
Signal2	16	Second chosen signal
Signal3	16	Third chosen signal
Signal4	16	Fourth chosen signal
Trigger Type	8	Trigger type for debugging process
Trigger Value	16	Trigger value for debugging process
pkg_ctl	16	Package type (not used)

TABLE 19: UNPKG_UDP_START_PULS INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
clk_1MHz	1	1 MHz clock
clk_100MHz	1	100 MHz clock
aresetn_1MHz	1	Resetn of corresponding 1 MHz clock
aresetn	1	Resetn of corresponding 100 MHz clock
start_100MHz	1	100 MHz start pulse (AXIS tlast signal)

TABLE 20: UNPKG_UDP_START_PULS OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
start_1MHz	1	1 MHz start pulse

TABLE 21: PKG_SAMPLES INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
clk_1MHz	1	1 MHz clock
m_axis_aclk	1	100 MHz clock
m_axis_aresetn	1	Resetn of the corresponding 100 MHz clock
framesize	12	Length of the AXIS data stream
send enable	1	Initialization to build the AXIS data stream
package_type	32	Package type of the data which will be sent with the UDP package. Currently package_type is set to 1 = data package
number_of_samples	32	Sample number to give the signal values a timestamp
Signal0	32	First chosen signal
Signal1	32	Second chosen signal
Signal2	32	Third chosen signal
Signal3	32	Fourth chosen signal

TABLE 22: PKG_SAMPLES OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
m_axis		Master interface for the AXIS data stream, which will be written into the RAM
initTF	1	Signal for the DataMoveCTL block to initialize the data transfer into the RAM

TABLE 23: DEBUGCOREMODULE INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
Input_Signals		Interface for the 300 input signals
Signal0	32	Input signal 1
…	…	…
Signal299	32	Input signal 300
clk_1MHz	1	1 MHz clock
resetn	1	Resetn of the corresponding 1 MHz clock
clk_1MHz_enable	1	Clock enable signal (constant 1)
c_NumOfSamples	32	Number of Samples for the debugging process
c_SampleRate	16	Sample rate for the debugging process
c_CMD	8	Command bits for different operation modes
c_TriggerType	8	Trigger type for the debugging process
c_TriggerValue	16	Trigger value for the debugging process
c_Signal0	16	First selected signal, also trigger signal
c_Signal1	16	Second selected signal
c_Signal2	16	Third selected signal
c_Signal3	16	Fourth selected signal

TABLE 24: DEBUGCOREMODULE OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
out_start_pkg1MHz	1	Signal to start the PKG_Samples block
debugged_NoS	32	Sample number for time stamp
debugged_Sig0	32	Debugged signal 1
debugged_Sig1	32	Debugged signal 2
debugged_Sig2	32	Debugged signal 3
debugged_Sig3	32	Debugged signal 4
db_CMD	8	Verification of received CMD bits
db_TT	8	Verification of received trigger type
db_TV	16	Verification of received trigger value
db_SR	16	Verification of received sample rate
db_NoS	32	Verification of received number of samples
db_Sig0	16	Verification of received chosen signal 1
db_Sig1	16	Verification of received chosen signal 2
db_Sig2	16	Verification of received chosen signal 3
db_Sig3	16	Verification of received chosen signal 4
db_SampleNum	32	Verification of received sample number
db_start	1	Verification of internal CMD_Start signal
db_triggered	1	Verification of the internal Start_Sampling signal
Resetn_1MHz	1	Resetn signal for the DataMoveCTL block

TABLE 25: SPLIT_CMD_BITS INPUT PORT AND INTERFACE

Port / Interface Input	Bits	Description
CMD_In	8	Command signal for the debugging settings

TABLE 26: SPLIT_CMD_BITS OUTPUT PORT AND INTERFACE

Port / Interface Output	Bits	Description
CMD_StartSampling	1	Initializes the debugging process
CMD_Reset	1	Resets submodules and stops the debugging process
CMD_Trigger	1	Enable trigger (post-trigger)
CMD_PreTrigger	1	Enable pre-trigger
CMD_NA1	1	Not accessed
CMD_NA2	1	Not accessed
CMD_NA3	1	Not accessed
CMD_NA4	1	Not accessed

TABLE 27: START_CONTROL INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
reset	1	Resets the CMD_Start signal
CMD_StartSampling	1	Initializes the debugging process

TABLE 28: START_CONTROL OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
CMD_Start	1	Starts the debugging process

TABLE 29: TRIGGER_CONTROL INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
Reset	1	Resets the Trigger_Control block
CMD_Start	1	Starts the waiting process for a trigger event
CMD_Trigger	1	Enables trigger
TriggerType	8	Selected trigger type
TriggerValue	16	Selected trigger value
Signal	32	Signal for trigger process

TABLE 30: TRIGGER_CONTROL INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Start_Sampling	1	Set, when trigger event happened

TABLE 31: SEND_CONTROL INPUT PORT AND INTERFACE DESCRPITION

Port / Interface Input	Bits	Description
Enable_NoSCounter	1	Enables the counter to increment
NumberOfSamples	1	Number of samples for the debugging process
Reset_Counter	1	Resets the counter

TABLE 32: SEND_CONTROL OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Reset	1	Is high, when number of samples is reached
Sample_Counter	1	Sample counter for time stamp
Sampling	1	Initializes the data transfer into the RAM. When the Pre-Trigger is active, it also initializes the read operation of the ring buffer with the data transfer into the RAM.

TABLE 33: SAMPLE_RATE_COUNTER INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
Reset_Counter	1	Resets the sample rate counter
start_counter	1	Enables the counter to increment
SampleRate	16	The value of the SampleRate signal is the compare value for the internal counter to determine the sample frequency.

TABLE 34: SAMPLE_RATE_COUNTER OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Enable_NoSCounter	1	Enables the Send_Control block with the sample counter

TABLE 35: RINGBUFFER INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
Sampling	1	Starts the read operation from the BRAMs
Reset	1	Resets the internal counters
WriteRam_Enable	1	Starts the write operation into the BRAMs
PreTrigger_Enabled	1	Enables the Pre-Trigger functionality
Sig_In1	32	Signal 1
Sig_In2	32	Signal 2
Sig_In3	32	Signal 3
Sig_In4	32	Signal 4

TABLE 36: RINGBUFFER OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Sig_Out1	32	Signal 1 for the AXIS data stream
Sig_Out2	32	Signal 2 for the AXIS data stream
Sig_Out3	32	Signal 3 for the AXIS data stream
Sig_Out4	32	Signal 4 for the AXIS data stream
Send_Enable	1	Enables the packaging process to build the AXIS data stream

TABLE 37: RINGBUFFERCTL INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
ReadRAM_Enable	1	Enables the read counter to increment
Reset	1	Resets the counters
WriteRAM_Enable	1	Enables the write counter to increment

TABLE 38: RINGBUFERRCTL OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
rd_addr	8	BRAM read address
wr_addr	8	BRAM write address

TABLE 39: RINGBUFFERSIG INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
Rd_Addr	1	BRAM read address
Wr_Addr	1	BRAM write address
WriteRAM_Enable	1	Enables write and read operations
Pre_Trigger_Enable	1	Selects the signal data from the BRAMs
Signal1_in	32	Signal 1
Signal2_in	32	Signal 2
Signal3_in	32	Signal 3
Signal4_in	32	Signal 4

TABLE 40: RINGBUFFERSIG OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Signal_Read1	32	Signal 1
Signal_Read2	32	Signal 2
Signal_Read3	32	Signal 3
Signal_Read4	32	Signal 4

TABLE 41: SIGNAL_SELECTION INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
Signals300	300 x 32	300 signal input vector, each signal has 32 bits
Signal_Sel	16	Signal selection for the debugging signal

TABLE 42: SIGNAL_SELECTION OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Signal_Selected	32	Selected signal for the debugging process

TABLE 43: PIPELINE INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
clk	1	clock
resetn	1	Resetn (active low)
sig_in	xx	Input signal (signal width can be chosen in the IP settings)

TABLE 44: PIPELINE OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Sig_out	xx	Output signal (signal width can be chosen in the IP settings)

TABLE 45: PIPELINE300 INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
clk	1	clock
resetn	1	Reset (active low)
Input_Signals	300 x 32	Input signal interface (300 signals with each 32 bits)

TABLE 46: PIPELINE300 OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Output_Signals	300 x 32	Output signal interface (300 signals with each 32 bits)

TABLE 47: CLKDOMAINCROSSING INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
clk_1MHz	1	1 MHz clock
resetn_1MHz	1	resetn (active low)
clk_100MHz	1	100 MHz clock
resetn_100MHz	1	resetn (active low)
in_signal_1MHz	xx	Input signal (signal width can be chosen in the IP settings)

TABLE 48: CLKDOMAINCROSSING OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Out_signal_100MHz	xx	Output signal (signal width can be chosen in the IP settings)

TABLE 49: SIG_GEN300 INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
clk_1MHz	1	1 MHz clock
resetn	1	resetn (active low)

TABLE 50: SIG_GEN300 OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
Gen_Signals	300 x 32 Bits	Output signal interface with 300 signals. Each signal has a width of 32 bits.

TABLE 51: AID IP CORE INPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Input	Bits	Description
clk_1MHz	1	1 MHz clock
clk_100MHz	1	100 MHz clock
resetn_1MHz	1	1 MHz Resetn (active low)
resetn_100MHz	1	100 MHz Resetn (active low)
MM2S_MEM_ADDR	32	Start memory address, where to read the data for the MM2S transfer
S2MM_MEM_ADDR	32	First start memory address, where to write the data for the S2MM transfer
S2MM_MEM_ADDR2	32	Second start memory address, where to write the data for the S2MM transfer
mm2s_data_length	12	Data length of the MM2S transfer
mm2s_startTF	1	Initialize MM2S data transfer from RAM to set up the debugging process with the parameters from the user-interface
s2mm_data_length	12	Data length of the S2MM data transfer
s2mm_number_pkg	16	Number of samples to collect for the UDP package, before the S2MM interrupt is set
s_axis_S2MM_S		AXIS interface for the S2MM transfer status
s_axis_mm2s_s		AXIS interface for the MM2S transfer status
s_axis_mm2s_d		AXIS interface for the MM2S data
Input_Signals	40 x 32 bits	Interface for the 40 or 300 input signals. The interface name is debugSig for 40 input signals and Input_Signals for 300 input signals.

TABLE 52: AID IP CORE OUTPUT PORT AND INTERFACE DESCRIPTION

Port / Interface Output	Bits	Description
m_axis_S2MM		AXIS interface for the S2MM data.
m_axis_S2MM_C		AXIS interface for the S2MM command data
m_axis_mm2s_c		AXIS interface for the MM2S command data
mm2s_finished_intr	1	Interrupt for a successful MM2S data transfer
s2mm_finished_intr	1	Interrupt for a successful S2MM data transfers.