ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **1**, Number 1, 2021, 117-126

Open-source SoC-based Off-the-shelf Industrial-Grade Low-Cost Logic Analyzer

Andrei GINDAC[1], Andrei-Edward POPA[1], Andrei-Alexandru ULMAMEI[1], and Calin BIRA[1]

[1]Dept. of DCAE, faculty of Electronics, Telecommunication and Information Technology, Univeristy Politechnica of Bucharest, Bucharest (Romania)
[1]E-mail: florin.gindac@stud.etti.upb.ro, andrei_edward.popa@upb.ro, andrei.ulmamei@upb.ro, calin.bira@upb.ro *

Abstract. This paper proposes a solution for back-to-back 200+ MHz 150-channel data acquisition based on Xilinxs Zynq-7000 family (dual-core ARM + Artix-class FPGA) with Linux Ubuntu rootfs and PYNQ software framework and application running in Jupyter notebook. A comparison is made between High Level Synthesis (HLS) and Hardware Description Language (HDL) versions of the device (performance, power consumption). It distinguishes itself from the state of art by providing a very good (MSa/s * channel) / USD value, in addition to large amounts of sample storage.

Key-words: SoC, FPGA, HLS, HDL, many channels, logic analyzer

1. Introduction

The logic analyzer is a tool utilized by the digital designer and has the advantage in the number of input channels (usually well over 8) as opposed to the analog-input digital oscilloscope (which typically has at most 4). The most advanced logic analyzers are capable of reading hundreds of channels and offer complex trigger conditions. Fast back-to-back data acquisition is essential in applications that involves high speed signal changes at input side. It is very important to have a high speed data acquisition without loss of any data. This paper discusses CPU-FPGA design architecture programmed in the Python language on top of C++ & FPGA firmware which involves a Zynq-7000 device (dual-core ARM + and Artix-class FPGA) and Ubuntu Linux operating system with an PYNQ-based [1] application running on top.

The integration for an ARM core (PS, programmed part) with the FPGA (PL, reconfigurable part) offers a good alternative to individual chips, due to the high datarate pipes available between the two, assuming they reside on the same die (or package). In paper [2] using the Xilinx

Zynq7000 SoC device, 3337 MB/s are achieved with 4 High Performance AXI datalinks, whereas using the CycloneV they achieved 2590 MB/s. According to them, Softcore SoC systems offer up to 9230 MB/s.

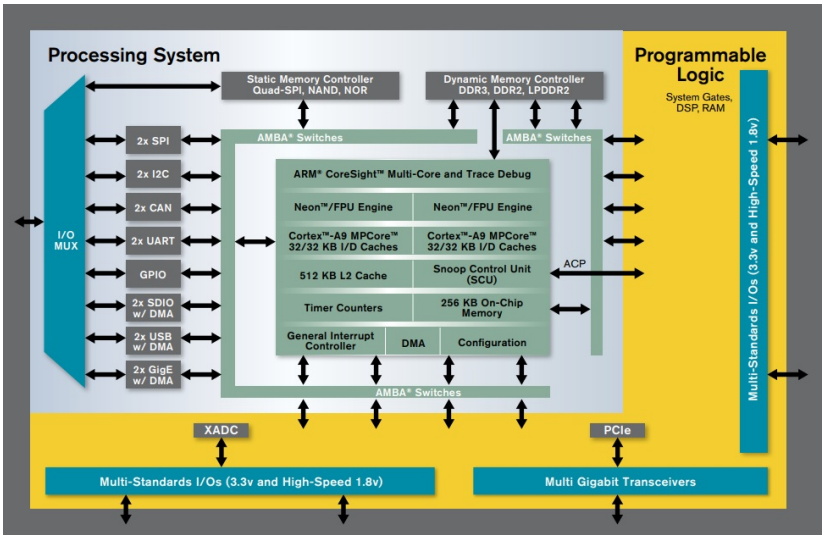

Fig. 1. Xilinx Zynq-7000 System-on-Chip.

2. Related Work

Cristian Zet [3] has implemented a 8-channel logic analyzer based on Altera's Flex10k20, with a maximum sampling frequency of about 25 MSa/s. It streams data to the PC via RS232/RS485 interface. The software application is written using Labview tools.

J. Luo et al. in [4] have used a CPLD-ADC-EZ_USB acquisition chain, where data is streamed over USB using the CY7C68013 FX2 mcu at a datarate of 40 MSa/s (8-bit) into the PC where it is decoded using a Borland C++ application.

L. Ehrenpreis, et al. [5] proposes the ChipSpy (a core similar to Altera's SignalTap or Xilinx's ChipScope Pro), based on Spartan3-series FPGA which tops at 200 MHz for low memory depth and width and 160 MHz for 256 bits and 4k samples; data transfer is done using a LPT port after the acquisition. The data lines are not exposed to the outside world, requiring the inspected design to reside in the same FPGA chip as the ChipSpy.

L. Weiping and S. Huan in [6] use an 8051 IP-core to interpret the commands sent by the PC running a LabView application which also implements complex trigger conditions.

A. D. Ioan and M. C. Ignat in [7] have used a VGA (1024x768 60Hz) IP-core and DAC output (3bpp) and 1K FIFO inside their FPGA to remove the need of an external PC. In addition they have used BERC as a filter. They use no software therefore argue that it provides an advantage (no expensive CPU is needed)

As far as commercial products are concerned, the Table 1 provides a comparison of commonly available devices with their features summarized. Given the trade-offs between number of channels and acquisition data rate, a new metric was introduced, the MSa/s * channels / USD.

Table 1. Commercial options

Hardware	**Software**	**Performance**	**System Cost ($)**	**MSa/s*ch**	**(MSa/s*ch)** per USD
Lattice iCE40HX1K [8]	Open-source SUMP2 [9]	16-ch 96 MSa/s	40	1536	38.4
USBee QX[10]	USBee Advanced Suite	24-ch 100 MSa/s 16-ch 200 MSa/s 8-ch 400 MSa/s	1495	3200	2.14
USBee SX[11]	USBee Free Suite	8-ch 24 MSa/s	130	192	1.47
Openbench[12]	sigrok[13]	16-ch 200 MSa/s	50	3200	64
Sysclk SLA5032 [14]	sigrok	32-ch 500 MSa/s	210	16000	76.2
Digilent Digital Discovery (Spartan-6) [15]	WaveForms [16]	8-ch @ 800 MSa/s 16-ch @ 400 MSa/s 32-ch @ 200 MSa/s	250	6400	25.6
Link Instruments LA-5580 [17]	Logic Analyzer FrontPanel	80-ch @ 250 MSa/s 40-ch @ 500 MSa/s	3500	20000	5.7
Link Instruments LA-55160	Logic Analyzer FrontPanel [17]	96-ch @ 500 MSa/s 160-ch @ 250 MSa/s 160-ch @ 250 MSa/s	7500	40000	5.3
Trenz 0703-6 carrier Trenz 0720-03-2IF SoM [18]	Open-source Python-based	152-ch 250 MSa/s	413	38000	92

3. Experimental setup

3.1. System description

The system used a Digilent Digital Discovery kit [15] for generating the stimuli data. The logic analyzer presented in this paper is based on a Trenz 0703-6 carrier and Trenz 0720-03 SoM [18]. The combination of carrier and SoM were chosen because of the high number of I/O available in 0.1 inch headers. The software ran on PS (programmable system, the ARM cores inside Zynq) on top of Ubuntu OS. The hardware description was implemented by both C++ code (using Xilinx's high-level synthesis) and Verilog HDL; multiple variants were tried and tested as follows below. In 3.2 we present a list of steps for the linux bring-up. In 3.3 multiple hardware implementations are described. All the source code may be found in [19]

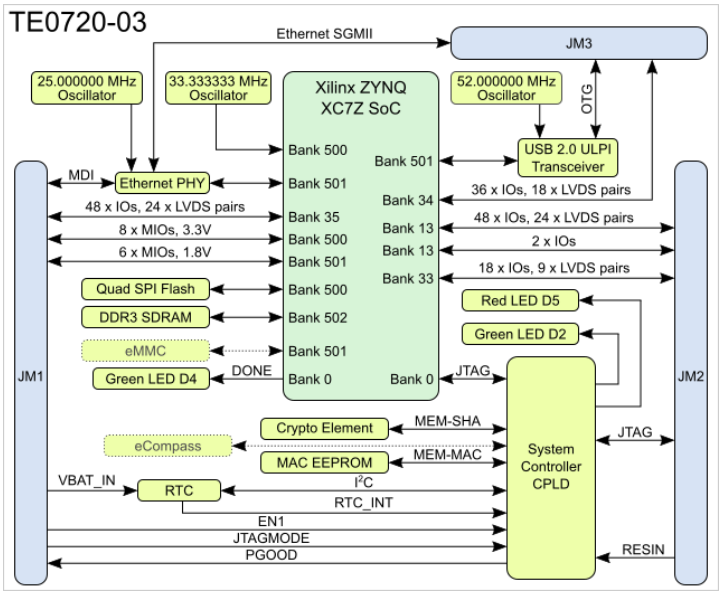

Fig. 2. TE0720-03 carrier board from Trenz Electronic providing 152 I/Os

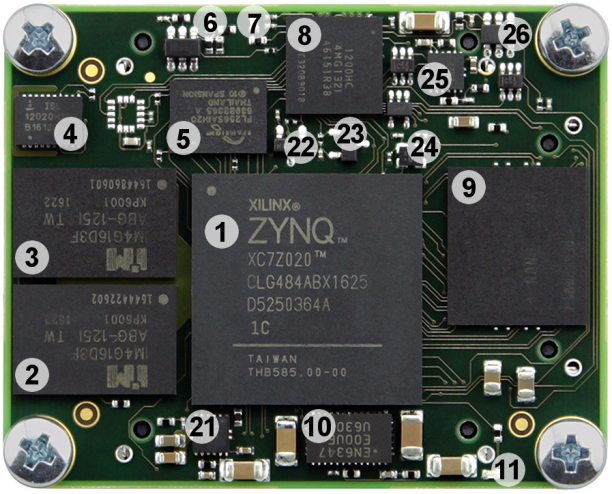

Fig. 3. The highly dense TE0720 SOM with the SoC(1), RAM (2,3) and QSPI memories (5)

3.2. Software setup

Installation of the Linux Image consists of four steps:

- creating the First Stage Bootloader;
- creating the U-Boot image;
- creating the Linux Kernel image;
- creating the Root File System;

The first stage bootloader is a program written in the RAM memory which initializes the processor and some peripherals. Since the board does not have Xilinx BSP, the authors had to create their own first stage bootloader (FSBL) file.

The flow of generating the FSBL file is:

- create a Block Design in Xilinx Vivado 2019.2
- describe the Zynq7000 coprocessors characteristics (eg. Fclk)
- use the project from PCB's manufacturer [19]
- creating the Root File System;

The authors used a project from Trenz Electronics [19], and the scripts for generating the FPGA bitstream file. It contained all the necessary data for programming the FPGA including the information needed by the Xilinx Vitis to create the FSBL. The Board Package consists of the FPGA bitstream for the Zynq7000 and the hardware support file (with .xsa extension). Those two files were used in Xilinx Vitis for creating a standalone C application, which was our first stage bootloader. In this program, we wrote the initialization code for the PS7 (Processing System). Once the .xsa file was generated, we used Xilinx Petalinux Tool in order to create the U-Boot image and the Linux Kernel image. The PetaLinux Tools offers everything necessary to customize, build and deploy Embedded Linux solutions on Xilinx processing systems [20].

U-Boot [21] is a open source bootloader for Embedded boards based on PowerPC, ARM, MIPS and several other processors, which can be installed in a boot ROM and used to initialize and test the hardware or to download and run application code.

The Linux Kernel image is a binary form of a Linux distribution (operating system) intended to be loaded onto a device.

Both the U-boot and the Kernel image source code come with a rather minimal graphical user interface (GUI) in the terminal, called menuconfig. Here we could choose what peripherals to further activate and what devices should be activated.

For the U-boot configuration, a serial console was used to communicate with the board (115.2kbps). The Zynq7000 has two separate UART interfaces and one of them was disabled, as recommended by Trenz.

For the Linux Kernel configuration, the authors activated the Userspace I/O platform driver for interrupt request handling. This was needed for the python software using the pynq framework. The mechanism works as follows: the interrupt request stops the execution of the processors normal task in order to communicate with the FPGA, for reading/writing data. For the device to be functional inside of the Linux OS, we had to add a node in the device tree file. After the PetaLinux project was built, the u-boot.bin, bitstream.bit and the image.ub files were created. Those, along with the hardware support file (.xsa extension) in Vitis, were utilized to create the first stage bootloader, using a script provided by Trenz Electronic. Once the FSBL was ready, the authors created a binary file (named BOOT.bin) containing the u-boot, first stage bootloader and the bitstream, used for booting the board.

The resulting files of interest are:

- BOOT.bin
- fsbl.elf
- *.bit (bitstream file)
- u-boot.bin (U-BOOT image)
- image.ub (kernel + device tree)

The board can be booted from SD card, USB flash drive or on-board QSPI memory. A script was created for detecting what boot device is available. When using SD card, a FAT16 partition

with at least 20 MB of free space, needs to be created.The rest of the card was formatted as EXT4 for RootFS. The rootfs used was Ubuntu 20.04.1. In the software image the authors added some useful tools like DNS solver, python 3.8.4, jupyter notebook, tornado, cffi, urllib3 etc.

After software installation was completed, the jupyter notebook server listened on port 9090. Here, one may upload his own overlays and python code. The authors wrote a small application to provide interface with the acquisition system (trigger, data transfers using DMA and data visualization in a text-mode interface)

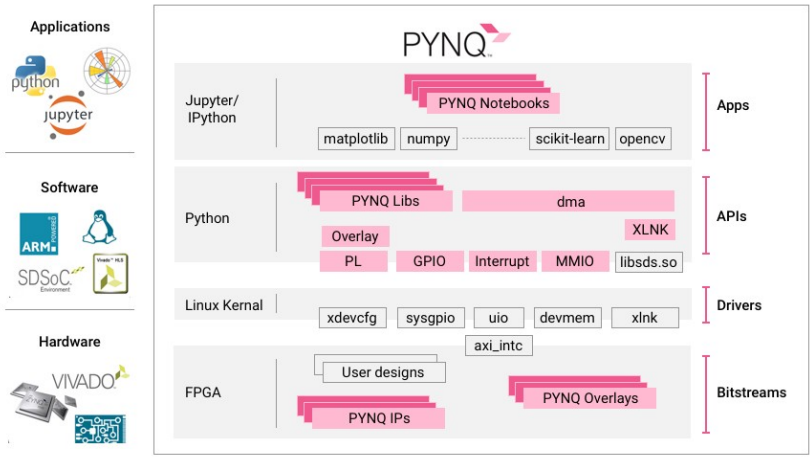

Fig. 4. The PYNQ software stack from top (high-level software) to bottom (digital design)

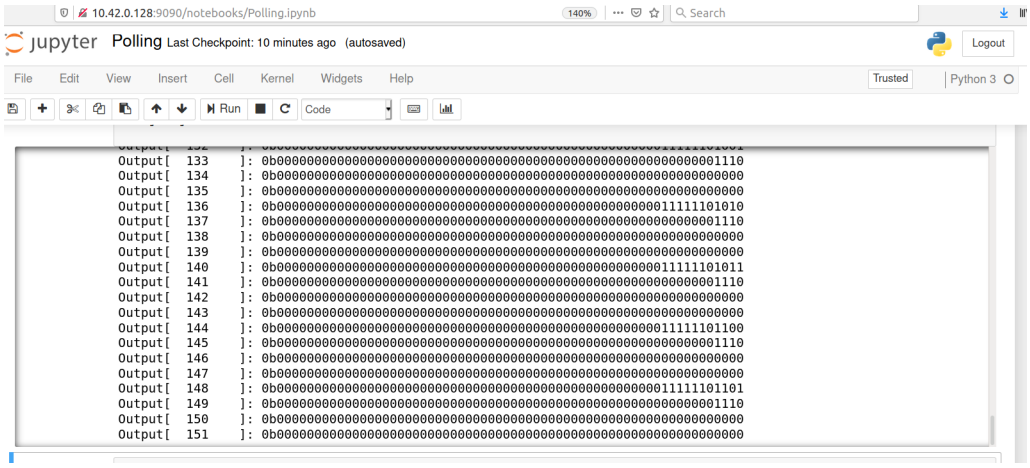

Fig. 5. Jupyter notebook running a data acquisition

3.2.1. Digital design

Two paths exist for describing the digital design: HLS (C++ code) and HDL (VHDL or Verilog). Both were explored below. Main task was to design an acquisition core that samples the data, adds a 64-bit timestamp and forwards it to the PS. The best implementation had a FIFO between the sampling core and DMA, and used AXI-Stream interfaces. The DMA transfered data from the FIFO via one 64-bit AXI HP interface.

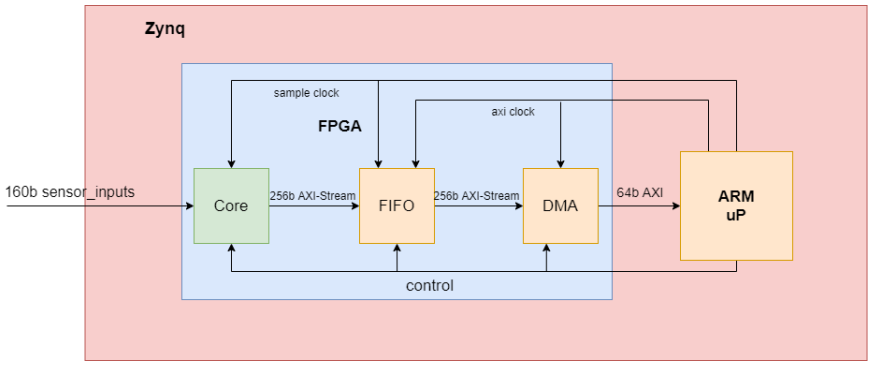

Fig. 6. Block schematic of the SoC from digital design pov

3.2.2. HLS implementations

The first implementation of the logic analyzer concept was done using the Vivado HLS tool. This allowed fast prototyping of digital functionalities using C/C++ code. It generated an RTL code from said code, which can be made to run more like a digital circuit by using pragma statements.

The HLS flow includes directives, which are methods by which one can further constrain the design. They can be used to prioritize resource use, latency or throughput. After the RTL simulation of the core is satisfactory, it can be exported as an IP and used in Vivado.

The attempt was to synthesize a circuit which would receive 152 inputs from the FPGA pins, and output a 256 bit output containing 64 bits for the time counter, 152 bits for the sensor inputs and 40 dummy bits. The HLS core would also contain logic for generating the valid signal, which is a logical OR, over all the sensor inputs.

The code used pragma statements (eg. #pragma DATAFLOW) to attempt to parallelize the counter and synchronization of the inputs. However it could not make the timestamp counter run independently, like an RTL counter. Instead, it acted like a hits counter, counting the number of times an input was triggered. This version was quickly abandoned, however, in favor of the first RTL implementation, which allowed for the aforementioned behavior of the counter.

The second implementation of the HLS core is called filter and, instead of trying to include the counter, it is considered an input and just passed along. At a block design level, it is paired with an AXI4-Stream FIFO IP which facilitates clock domain crossing, and an AXI DMA IP, which allows communication with the ARM microprocessor. The second HLS core utilizes the #pragma ap_ctrl_none statement, which removes block- level I/O protocols of the function and makes it run more an RTL code. The integration at the top is the place where most improvements have been done, by adding a 64 bit counter IP, an AXI4-Stream FIFO and AXI4 DMA, which enable communication with the processing system IP.

This implementation was able to go up to 200MHz, but its throughput was limited by the fact that the HLS core has to reset after each transaction, thus practically having half the frequency.

3.2.3. HDL implementations

The first RTL implementation had several functionalities done inside the core. These are:

- synchronization
- time stamping
- clock domain crossing
- AXI4-Stream master interfacing

Each of these functionalities are inside their own modules. The synchronization module is a basic module which just synchronizes all 152 inputs from the sensors. It also generates the valid signal for the saving the data, which is an or of all the inputs. The timestamp module is just a basic 64-bit ripple counter which represents the time-base for each hit. The clock domains were crossed using an asynchronous FIFO whose pointers are represented by 2 grey counters. These counters are synchronized into each others clock domain using dual flip-flops and these synchronized counters are used to ensure that the read pointer does not go ahead of the write one. This module would also transmit a valid signal to the AXI4-Stream master interface, which, in turn, begins a new transaction. The transaction start signal whenever the read pointer had not caught up to the write pointer. This pointer synchronization between clock domains added some delay and an important limitation to the maximum frequency usable by the circuit.

The top module contains a AXI4 FIFO which attempted to bridge the communication gap between the stream and normal AXI4 protocols. This attempt to integrate the asynchronous FIFO in the design yielded a circuit which could only be synthesized at frequencies up to 150MHz. Due to the fact that attempts to connect to the core via software failed, and the synthesis of the RTL asynchronous FIFO would sometimes yield timing errors, a HLS approach was tried which proved much more successful and allowed for deployment on the FPGA. The throughput of the AXI4 FIFO is also smaller than that of the AXI4 DMA, making it even less suited as a solution for this problem.

RTL-II and RTL-III versions. To surpass the limitations of the previous implementation, the asynchronous FIFO was moved outside of the core and replaced with an IP. As such, the core became primarily a tool for packing the timestamp with the synchronized inputs and sending them via AXI4-Stream toward an AXI4-Stream FIFO where the data crosses in the main AXI4 clock domain. Also, this implementation does not suffer from the same issues as the second HLS one, because the core does not require one extra clock cycle to reset. By removing the asynchronous FIFO, the design can be synthesized to approximately 175MHz. The relevant addition at this stage are to the block design. It has been improved in numerous ways by adding the above-mentioned FIFO, an AXI DMA, to facilitate access to the FIFO, delay lines for timing balances and AXI4-GPIO controllers for testing purposes. The integration is very much alike the one used in the second HLS one, but lacks the external counter. Another design step was taken after this one. The integration has been kept primarily the same, the only exception being the counter, which was moved into the block design to utilize the Xilinx IP for improved performance. Another change in the third RTL implementation is to the core. It rebuilds the OR-tree customly to fit the Xilinx slices, using only 6 inputs for LUT and FF pair usage maximization. This or-tree is pipelined to increase the frequency even further, up to 250MHz. It also adds burst functionality for the AXI4-Stream, which increases the potential throughput of the application.

To sum up all digital design experiments:

Parameter	HLS-II	RTL-I	RTL-II	RTL-III
FMAX (MHz)	200	150	175	250
POWER (W)	1.44	1.78	1.45	1.56
Memory (Mb)	0.52	4.19	0.52	1.04
Memory (Sa)	2k	16k	2k	4k
Max generated data (Gbps)	25.6	38.4	44.8	64

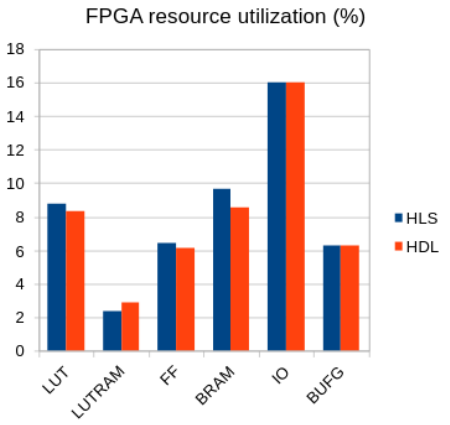

Fig. 7. Comparison between various RTL (HDL) and HLS implementations (in terms of FPGA resources and capabilities)

- HLS I was the first attempt, written in C++ for Vivado HLS. Though not integrated, the HLS tool estimated it could go to approximately 280MHz. Its main flaw and reason for which I moved to HLS I was the inability to instantiate a simple counter using the HLS tools.
- RTL I was the safe solution afterwards, where we tried to incorporate as much as possible of the functionality in the core. RTL I was written in SystemVerilog. Its main flaw was the relatively modest frequency it could reach, only 150MHz.
- More reliance on Vivado IPs to do some of the work which in RTL was done manually. It was more of an optimization of RTL I with lessons learned from HLS. Thus, the frequency went up 175MHz.
- HLS II and RTL III were tweaked for memory and sampling frequency, and final best results were: HLS II at 200MHz and RTL III at 250MHz

4. Conclusion

Vivado HLS lacks the ability to implement true circuit-like behavior. It is optimized for high frequency. HLS implementation was not functionally identical to the HDL implementation: due to how HLS is in concept, a FSM (finite state machine) is generated in hardware, and gets reset every N samples: while somewhat controllable, this reset implies an 1 clock-cyle with the system offline, unable to see variations of inputs. From this point of view, the HDL implementation is the only one that provides a true back-to-back data acquisition solution.

PYNQ bypasses the time-consuming part of interfacing the FPGA cores and on-board processors, allowing quick development.

The best HDL implementation achieved 250 MHz whereas the best HLS one, only 200.

Acknowledgment This work was supported by a grant of the Romanian Ministry of Research and Innovation, CCCDI UEFISCDI, project number: PN-III-P1-1.2-PCCDI-20170839 /19PCCDI2018 within PNCDI III

References

[1] *PYNQ open-source project from Xilinx* Available at: http://www.pynq.io/

[2] M. Gobel, "A Quantitative Analysis of the Memory Architecture of FPGA-SoCs" *Applied Reconfigurable Computing*, 2017, Springer International Publishing, 978-3-319-56258-2

[3] C. Zet and C. Fosalau, "FPGA Based Logic Analyzer," 2018 *International Conference and Exposition on Electrical And Power Engineering (EPE)*, Iasi, Romania, 2018, pp. 0335-0340, doi: 10.1109/ICEPE.2018.8559948.

[4] J. Luo, Y. Xia, A. Li and C. Yang, "The design and realization of DSRC logic analyzer based on USB," *2010 2nd International Conference on Future Computer and Communication*, Wuhan, China, 2010, pp. V3-37-V3-40, doi: 10.1109/ICFCC.2010.5497671.

[5] L. Ehrenpreis, P. Ellervee and K. Tammemae, "Open Source On-Chip Logic Analyzer for FPGAs," *2006 International Biennial Baltic Electronics Conference*, Tallinn, Estonia, 2006, pp. 1-4, doi: 10.1109/BEC.2006.311070.

[6] L. Weiping and S. Huan, "Design of a Virtual Logic Analyzer Based on FPGA," *2016 Sixth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*, Harbin, China, 2016, pp. 38-42, doi: 10.1109/IMCCC.2016.206.

[7] A. D. Ioan and M. C. Ignat, "FPGA autonomous logic analyzer using innovative BERC filter optimization," *2015 7th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, Bucharest, Romania, 2015, pp. E-39-E-46, doi: 10.1109/ECAI.2015.7301150.

[8] *iCEstick Evaluation Kit* Available at: http://www.latticesemi.com/view_document?document_id=50701

[9] *SUMP2 an open-source software and hardware logic analyzer* Available at: https://github.com/blackmesalabs/sump2

[10] *USBee QX logic analyzer* Available at: https://www.usbee.com/qx.html

[11] *USBee SX logic analyzer* Available at: https://www.usbee.com/sx.html

[12] *Openbench Logic Sniffer* Available at: https://sigrok.org/wiki/Openbench_Logic_Sniffer

[13] *Sigrok: portable, cross-platform, Free/Libre/Open-Source signal analysis software suite* Available at: https://sigrok.org/

[14] *The Sysclk SLA5032 USB-based, 32-channel logic analyzer* Available at: https://sigrok.org/wiki/Sysclk_SLA5032

[15] *Digital Discovery: Portable USB Logic Analyzer and Digital Pattern Generator* Available at: https://reference.digilentinc.com/reference/instrumentation/digital-discovery/reference-manual/

[16] *WaveForms virtual instrument suite* Available at: https://store.digilentinc.com/digilent-waveforms/

[17] *LA-5000 Logic Analyzer, Pattern Generator* Available at: https://www.linkinstruments.com/logana5.html

[18] *Trenz SoC Module with Xilinx Zynq XC7Z020-2CLG484I* Available at: https://shop.trenz-electronic.de/en/Products/Trenz-Electronic/TE07XX-Zynq-SoC/TE0720-Zynq-SoC/

[19] *Presented system repository* Available at: https://gitlab.dcae.pub.ro/research/ifin_scintilators/-/tree/TE0720/

[20] *Xilinx PetaLinux Tools* Available at: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[21] *U-Boot: boot loader for embedded* Available at: https://github.com/u-boot/u-boot