Open-source SoC-FPGA Platform for Signal Processing

1st Matías Javier Oliva
Grupo de instrumentación Biomédica Industrial y Científica
Universidad Nacional de La Plata
La Plata, Argentina
matias.oliva@ing.unlp.edu.ar

2nd Pablo Andrés García
Grupo de instrumentación Biomédica Industrial y Científica
Universidad Nacional de La Plata
La Plata, Argentina

3rd Enrique Mario Spinelli
Grupo de instrumentación Biomédica Industrial y Científica
Universidad Nacional de La Plata
La Plata, Argentina

4th Alejandro Luis Veiga
Grupo de instrumentación Biomédica Industrial y Científica
Universidad Nacional de La Plata
La Plata, Argentina

***Abstract*—Systems known as SoC-FPGAs have experienced a growing popularity in recent years. This devices integrate field programmable gate arrays with elements such as microprocessors, PLLs and embedded memory blocks. The advantages of this type of systems are clear: great reconfigurability, performance, and energy efficiency, but they come with an negative side: programming and optimizing the applications that use them remains a long and complicated process. In particular, real-time signal processing at high frequencies is an application that can clearly benefit from the advantages of SoC-FPGAs, but the complex workflow assosiated with them usually prevents the designers from taking advantage of its capabilities. In this work, an open source SoC-FPGA platform, specifically intended for signal processing is presented, with the aim of alleviating this workflow. The platform structure is described, specifying the places where the designer may implement their algorithms, and then its operation is demonstrated by acquiring a signal at a maximum sampling frequency of 65 MHz and passing it through a 32th order FIR filter, verifying that the it meets it's expected theoretical response. The whole system can operate at a maximum frequency of 85 Mhz, has a latency of 16 clock cycles, and uses less than half of the resources of a Cyclone V device.**

***Index Terms*—SoC-FPGA, signal-processing, open-source, filtering .**

I. Introduction

In recent years, there has been a growing demand of computational capacity. This demand has motivated the development of new architectures and computational systems among which solutions based on field programmable gate arrays (FPGAs) stand out. These devices are highly re configurable, with high performance and energy efficiency, and the chips that integrate them have evolved to include different sub-systems that complement their capabilities. In this process, systems known as SoC-FPGAs have emerged, which integrate an FPGA with embedded memory blocks, phase locked loops (PLLs), digital signal processing blocks (DSPs) and even microprocessors in a single chip [1] [2].

In particular a SoC-FPGA system with an embedded microprocessor is an interesting platform for implementing high speed signal processing systems, with the FPGA in charge of the real-time processing, including the control of the analog signal acquisition and generation systems, and the microprocessor controlling the operation and the user interface. This guarantees complete control of the signal at clock transfer level, while keeping the user interface friendly and versatile.

The usual workflow when designing an application on a SoC-FPGA system begins with hardware design at register transfer level (RTL) in some hardware description lenguage (HDL) like Verilog or VHDL, with the assistance of some simulation tool for its verification. Special care must be taken when elements such as PLLs or DSP blocks are needed, since the HDL must be written correctly for the compiler to be able to infer them. Compilers provided by different manufacturers (Intel-Altera's Quartus or Xilinx's Vivado, for example) are then used to generate the "bitstream" needed to program the FPGA, which has to be tested experimentally to solve possible problems that have not appeared in simulation. Additionally, in SoC-FPGAs that include a microprocessor, the programming of this processing element must be done independently, with code in some high-level language such as C, C++, Python, etc. This code must be compiled for the target microprocessor, either by trans-compiling it with vendor-supplied tools, or by compiling it natively on the microprocessor. Finally the whole system must be verified. This long and complicated process requires designers highly skilled in digital design and in the particular architecture of the target platform [3].

In order to alleviate this workflow, High level synthesis tools (HLS) have emerged [4] [5] [6] [7]. This tools allow the generation of HDL code from C/C++/OpenCL code. Although these languages generate sequential programming codes, they allow parallelized algorithms to be implemented using computation directives. This greatly simplifies the design of algorithms, but does not simplify the rest of the steps involved in the system's design.

In this article an architecture of a SoC-FPGA system, specifically intended for signal processing, will be described.

The design includes digital an analog signal inputs, a model for the processing stages of the system, and the means to control the signal flow and retrieve the results of the processing. Its objective is that the programmer should only concentrate on the application of the signal processing algorithms, either writing HDL or using HLS tools, without worrying about the integration of the different sub-systems. In order to test the system a 32th order finite impulse response (FIR) filter with re configurable coefficients was implemented.

The design is open source, licensed under the terms of the MIT license, and available at [8]. It was implemented on an Intel-Altera Cyclone V SoC-FPGA [9], mounted on a DE1-SoC development kit, provided by Terasic [10]. Verilog was used for HDL programming, with the addition of some free licenced Intel-Altera intellectual property (IP) blocks, and tools written in C/C++ and C# were developed for the control of the operation.

II. System description

The proposed design is schematically shown in Fig. 1.

A. Control Stage

The control stage includes the hardware designed in the FPGA, which contains elements to configure the processing operation, control its speed and flow, and collect available results, and the control element itself. This can be a microprocessor integrated on the SoC or a "soft" processor, implemented in the FPGA fabric, depending on the availability in the target platform and the designer's needs. For the latter, the free version of the NIOS 2 processor, a 32-bit RISC processor optimized to save area on the FPGA [11], was implemented, and to communicate the design with the external processor, if available, the "Lightweight-axi-bus" was used. In the Cyclone V platform this processor is an ARM-Cortex A9. Hardware abstraction layers (HAL), written in C/C++, are provided for both modes of operation.

The main system's clock is generated trough a PLL, available inside the Cyclone V chip. Two IP blocks provided by Intel-Altera are used to instantiate it: "Altera PLL" [12] and "Altera PLL reconfig" [13]. These blocks allow to generate, from a 50 MHz clock, one with a frequency between 1 and 65

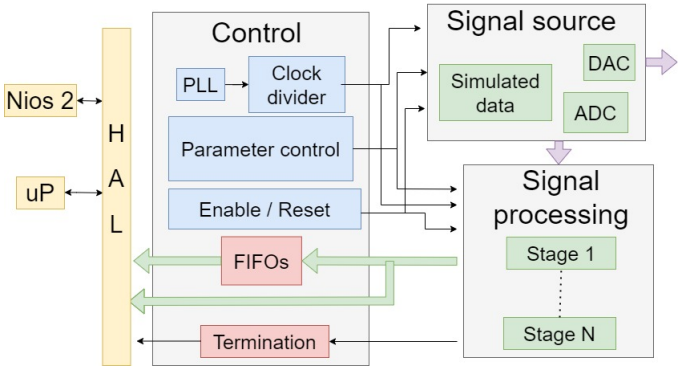

Fig. 1. Design's general structure.

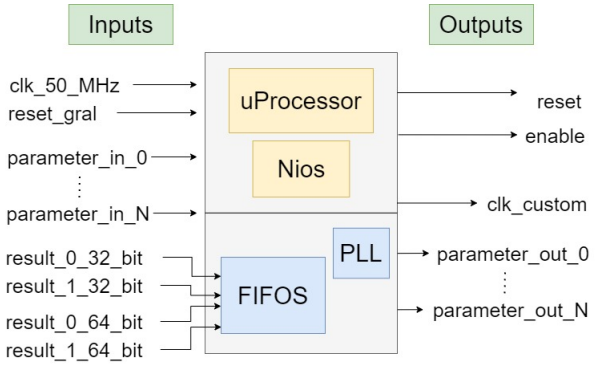

Fig. 2. Control module.

MHz. Additionally, a clock divider is provided, to implement lower frequencies. The enable, reset, and termination signals are used to control the flow of the process.

To parameterize the operation at run time a parameter control stage is included. Some examples where this may be useful are when the user wants to control the number of cycles during which a signal is integrated, or the coefficients of a filter.

The results of the processing enter the module in 32 or 64 bit formats, and are stored in First in first out (FIFO) memories. In addition to the data to be stored, the processing logic must include a "data_valid" signal, which is set high on every clock cycle that the result is valid. This scheme is used at all stages of processing to ensure its correct synchronization, and is known in the Intel-Altera documentation as an "Avalon Streaming" interface. It's use is not limited to Altera's hardware.

FIFO memories were implemented using Altera IP blocks, with an "Avalon Streaming" input interface, and an "Avalon Memory Mapped" output interface [14]. This input interface allows directly adapting the memories with the designer's own logic, as long as it complies with the rules described earlier. This text will not delve into the "Avalon Memory Mapped" interface, other than to say it is the one that allows the control element to correctly read the memories. The interested reader can find more about this interfaces in [15].

B. Signal source stage

The signal source stage, represented on Fig. 3, controls the input signals for processing. These can be digital, for testing proposes, or analog, incoming from some analog to digital converter (ADC).

1) Digital Signal: The digital sinusoidal signal is generated from a look-up table, and optionally contaminated with uniform noise, through a pseudo-random sequence. The module provides a signal sample on each rising clock edge on which the enable signal is high. The designer can configure the number of points per cycle used to generate the sinusoid, the method to obtain the pseudo-random sequence that simulates the uniform noise, and its amplitude.

The pseudo-random sequence can be generated using an algorithm called linear feedback shift register (LFSR) [16],

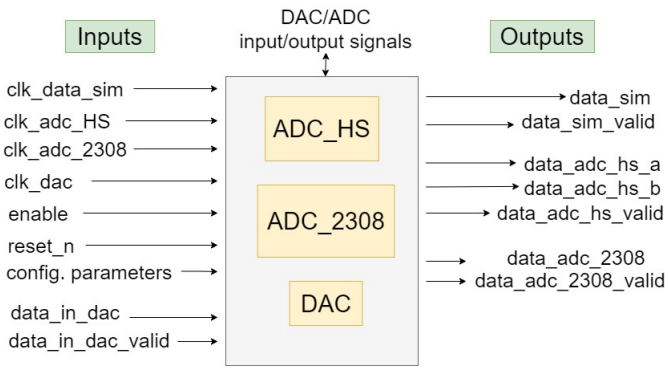

Fig. 3. Signal source module.

or by a typical linear congruential generator, given by the equation 1. In this equation the operator % represents the modulus operation, and the parameters used are: $c = 1$, $a = 69069$, $m = 2^{32}$, which are ones used by old versions of the GNU C library (glibc) [17].

$$N_{i+1} = (aN_i + c)\%m \tag{1}$$

Once the pseudo-random sequence is generated, it is scaled according to the requested noise amplitude and added to the sinusoidal signal. In this way sinusoidal signals with different levels of signal-to-noise ratio can be generated. This can be useful to test the immunity of the different algorithms against noise, for example.

2) Analog Signal: To obtain analog signals for further processing two different ADC drivers are included. The design allows them to be operated at different frequencies through the clock circuitry and provides synchronization with other modules through an Avalon Streaming Interface.

The first one is the ADC LTC2308 [18], a 12-bit resolution and 500kHz maximum sample rate with eight multiplexed channels, usually included in Terasic platforms. Its operation is through an SPI bus, which needs a clock of 40 MHz of frequency at maximum. The sampling frequency of this module can be configured by the designer. The output signal, "data_adc_2308", includes its respective "data_adc_2308_valid" signal.

The other one is an AD9248 [19], a 14-bit resolution ADC with 65 Msps maximum sample rate and two independent channels, included in the "High-speed A/D and D/A Development Kit" platform, also from Terasic [20] .This ADC is controlled by a parallel interface, so it can operate at the clock provided by the "clk_adc_hs", which can be connected to the system clock generated on the control stage. The output signals "data_adc_hs_a" and "data_adc_hs_b" also include their respective "data_adc_hs_valid" signal.

Additionally, the system provides the driver for a digital to analog converter: The AD9767 [21], which is a 125 Mmps, 14-bit resolution converter. This converter is operated through a parallel interface, so it can be operated directly at "clk_dac" speed. The controller included in the design converts to analog

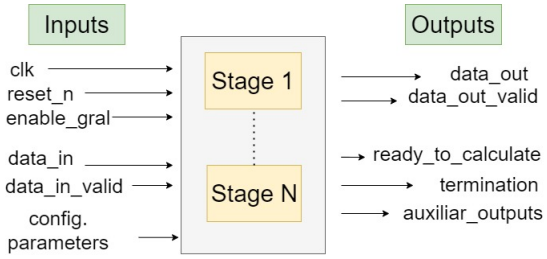

Fig. 4. Signal processing module.

each sample that enters the module through the "data_in_dac" bus, as long as the "data_in_dac_valid" signal is high. This can be used for example to generate a waveform using an internally stored lookup table, or to convert back to analog the results of the processing on the incoming signal.

If other ADCs are to be connected to the system the designer's logic has to provide two signals: a "generic_adc_data" bus, and a "generic_adc_data_valid" wire, guaranteeing that the driver provides a valid sample in the first one at each clock cycle where the latter is in high state.

C. Signal processing stage

Regardless of which signal source is selected, data enters the signal processing stage one sample at a time for each clock cycle that "data_valid" is high. This stage, which has the input and output interface shown in Fig. 4 is where the programmers may implement their algorithms. These can be further subdivided on different sub-stages or sub-systems, with each one working as an enablement for the next. The stages can be parameterized at execution time, through the different configurable parameters, and once the processing is finished they must set the completion signal high to inform the control module of the availability of the results. An extra signal, represented as "ready_to_calculate" in Fig. 4, tells other modules that the signal processing stage is ready to start receiving the data samples. This is useful when this module has to update parameters, or clean internal buffers before starting to process the signal, for example.

III. Signal processing example

In order to test the system a setup like the one shown in Fig. 5 was implemented. In this configuration the signal is generated with the SR865 lockin and acquired by the AD9248 at a configurable sampling frequency, up to 65 MHz. Then it passes through a 32th order FIR filter, with configurable coefficients. This filter follows the classic FIR filter equations, shown in equation 2 where the b_i are its coefficients, which enter the processing module as 16 bit integer numbers through the parameter control module.

$$Y_n = \sum_{i=n-M}^{n} b_i x_i \tag{2}$$

The microprocessor implements a C++ program which controls the operation and runs directly on its operating system:

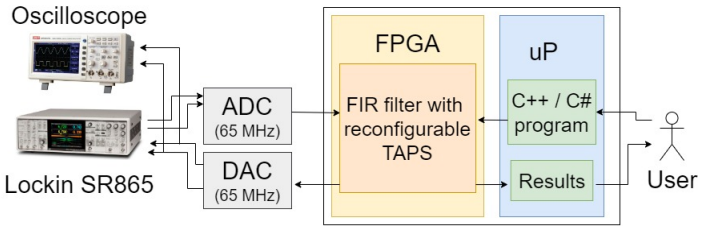

Fig. 5. Testing setup.

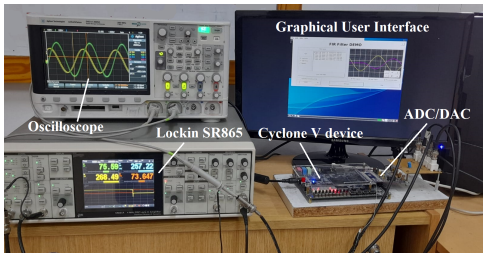

Fig. 7. Testing Setup.

a Linux with an Ubuntu distribution. This program can be executed from a terminal connected to a personal computer through a serial interface, or directly from the SoC-FPGA, if a monitor and keyboard is connected. For this configuration, which is the one selected for this demonstration, a graphical user interface (GUI) was designed in C#, and executed through the Mono implementation of the .Net framework [22]. This GUI, shown in Fig. 6 allows the user to easily set up the operation, and implements named pipes to configure the FPGA through the C++ HAL.

Using this program the user can configure the filter coefficients and the sampling frequency. In this way the same system can be used to implement different type of filters, at different cut-off frequencies. Once processed, the signal is fed to the DAC, in order to see the input and output signals together on an oscilloscope, and also stored on the FIFO memories, so the user can read them directly on his or her computer screen. The GUI uses this information to plot fragments of the signal, so the user can verify the operation without further equipment. Finally, the SR865 lockin is used to measure the amplitude of the output analog signal, in order to verify the operation of the whole system.

The filter coefficients for this demonstration were selected to implement low-pass and high-pass filter of different normalized cutoff frequencies ($0 < \omega < 1.0$). The coefficients enter the processing module as 16 bit integer numbers, through the parameter control module. To convert the coefficients provided by some signal processing toolbox (Python's numpy or Matlab, for example), one simply has to multiply the coefficients by 2^{16} and then round the number to the nearest integer. The final cutoff frequency of the filter depends on the selected ω and the sampling frequency, as shown in equation 3.

$$f_{cut} = \omega f_{sampling}/2; \tag{3}$$

A photo of the testing system is shown in Fig.7.

IV. Results

A. Resource utilization

The resource utilization of the proposed system depends on the election of the control element. If the microprocessor is to be used considerable memory blocks can be saved, but more logic elements are needed. On the other hand, if the Nios processor is selected, the system needs a significant amount of extra memory blocks, as the Nios 2's program memory is implemented directly on them. Finally, for the FIR demonstration, the system needs many DSP blocks, which are used to implement the multiplications efficiently. It's operation could, however, be replaced by multipliers implemented with logical elements, with an important reduction of time efficiency and an increase in area. This could be a good choice for devices with less embedded multipliers. The resource utilization summary for each mode of operation is shown in Table I.

B. Timing measurements

For each system's configuration the maximum achievable clock frequency was calculated using the tools provided by Intel Altera. In all the cases the maximum required frequency for this demonstration (65 MHz) was achieved. In the cases where the FIR filter is not implemented the achievable frequency is limited by the few Altera's IP used in the design. In the cases were the FIR filter is implemented it's the filter who limits the frequency. If a greater speed is to be achieved other filter architectures must be considered.

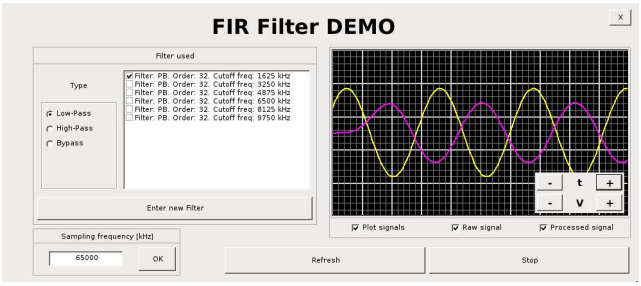

Fig. 6. Graphical User Interface.

TABLE I
Resource Utilization

Resource	With μP	With NIOS	Full system
LE (in ALMS)	3875 (12%)	2212 (7%)	6992 (22%)
Registers	5937	3170	11282
Memory blocks	262272 (6%)	1473536 (36%)	1342592 (33%)
RAM blocks	30 (8%)	195 (49%)	183 (46%)
DSP blocks	0	0	67 (77%)
Pins	157 (34%)	85 (19%)	157 (34%)
PLLs	1 (6%)	1 (6%)	1 (6%)

*Percentages calculated for Cyclone V 5CSEMA5F31C6N device
*Full system: μP + NIOS + 32th order FIR filter

TABLE II
ACHIEVABLE CLOCK FREQUENCY

Nios processor	μP	FIR filter	Achievable Frequency
✓	-	-	115 MHz
✓	-	✓	82.31 MHz
-	✓	-	97.82 MHz
-	✓	✓	87 MHz
✓	✓	✓	85 MHz

As the data flows through the system it is registered in several modules, implementing a pipeline. This pipeline allows the maximum frequency of the clock to reach the levels described earlier, but produces a latency in the output signal. This latency can be estimated by following the signal path, schematically shown in Fig. 8. Firstly the signal is acquired by the AD9248, which has a pipeline delay of 7 clock cycles, according to its data-sheet [19]. Then it is registered on the ADC driver module, which takes 2 clock cycles. The FIR filter registers the signal, then calculates the 32 required multiplications and finally sums the results. This produces a pipeline delay of 3 clock cycles. Then the DAC driver registers the output data and conditions the signal, in a total amount of 3 clock cycles. Finally the AD9767 latchs the output, 1 clock cycle later. The signal path sums for a total amount of 16 clock cycles.

To measure the latency in the signal the FIR filter was bypassed, by tuning all its coefficients to 0 except of the first one. This produces no change in the incoming signal, except the delay produced by the system's pipeline. Then the time distance between the input and output signal was measured with an oscilloscope, as shown in Fig. 9. For a sampling frequency of 10 MHz a delay of approximately 1.6 μS was obtained, which is consistent with the estimated 16 clock cycles.

C. Filter's response

With the setup described earlier, a low-pass and a high-pass filter were implemented and tested. Both filters were designed with a cutoff frequency of $\omega = 0.05$, and a sampling frequency of 1 MHz was selected. With these parameters a cutoff frequency of 25 kHz is achieved, as expected from equation 3. The coefficients selected for the filters were obtained from Python's "numpy" signal processing toolbox.

With this configuration the amplitude of the transfer functions $\mid H(f) \mid$ of both filters was measured with the SR865,

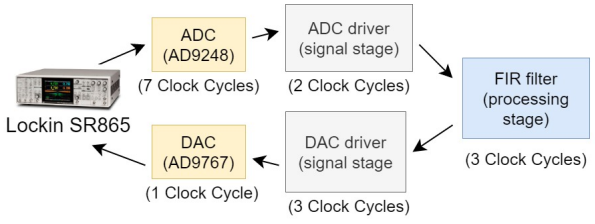

Fig. 8. Signal path.

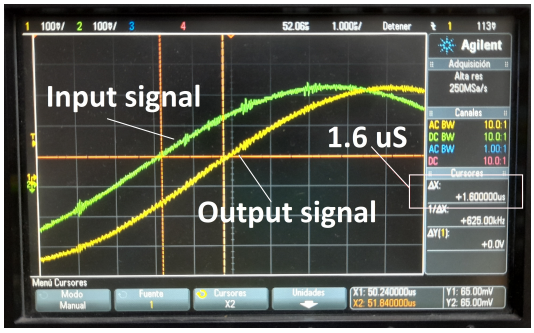

Fig. 9. Timing delay measurment.

and then compared with the theoretic transference of the filters. The results are shown in Fig. 10, and Fig. 11.

CONCLUSIONS

In this paper a SoC-FPGA design intended for signal processing was developed, providing a well structured design that developers can follow. The result is an open-source system, with options to adapt it to different Intel Altera's SoC-FPGAs, and with core concepts that are transferable to other vendors architectures.

The design was tested implementing a 32th order FIR filter that operates in real time with high frequency signals, a system with speed and data throughput requirements restrictive for most traditional microprocessors. The filter's transfer characteristic was measured using a SR865 lock-in, verifying that it meets the expected theoretical response.

We believe that this work has a great number of industrial and educational applications, as it simplifies the heavy workflow usually associated with these state-of-the-art embedded systems. The future work on this subject will be related with the design of other signal processing modules, with the objective of improving this open-source signal processing platform.

REFERENCES

[1] Yang, H., Zhang, J., Sun, J. et al. "Review of advanced FPGA architectures and technologies". J. Electron.(China) 31, 371–393 (2014). doi: 10.1007/s11767-014-4090-x

[2] Monmasson, E. and Cirstea, Marcian. "FPGA Design Methodology for Industrial Control Systems—A Review". Industrial Electronics, IEEE Transactions on. 54. 1824 - 1842. doi: 10.1109/TIE.2007.898281 (2007).

[3] Huang, Sitao; Wu, Kun; Jeong, Hyunmin; Wang, Chengyue; Chen, Deming and Hwu, Wen-mei. "PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow". IEEE Transactions on Computers. 70. 1-1. doi: 10.1109/TC.2021.3123465 (2021).

[4] Intel Corporation. "Intel® High Level Synthesis Compiler: User Guide". [Online] https://www.intel.com/content/www/us/en/docs/programmable/683456/22-3/pro-edition-user-guide.html (accessed on November 2022).

[5] AMD- Xilinx. "Vitis High-Level Synthesis User Guide (UG1399)". [Online] https://docs.xilinx.com/r/en-US/ug1399-vitis-hls (accessed on November 2022).

[6] Intel Corporation. "SDK Intel® FPGA para OpenCL" [Online] https://www.intel.la/content/www/xl/es/support/programmable/support-resources/design-software/opencl-support.html (accessed on November 2022)

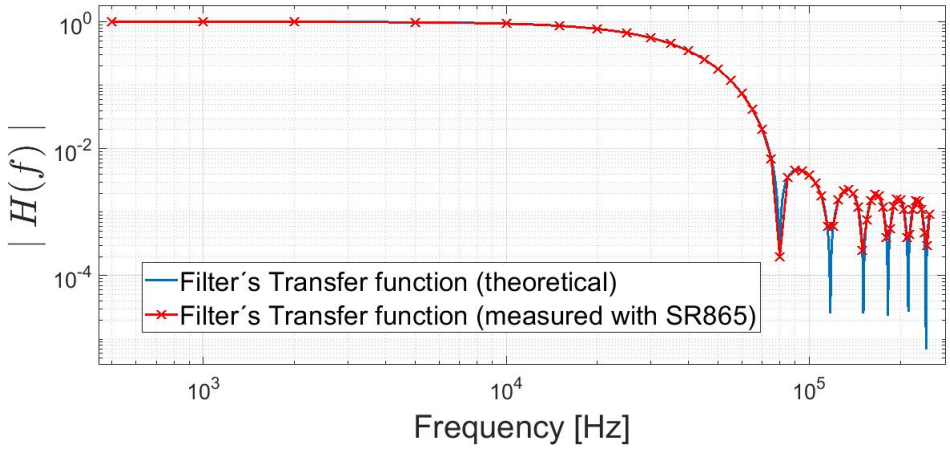

Fig. 10. Low-pass filter's transference.

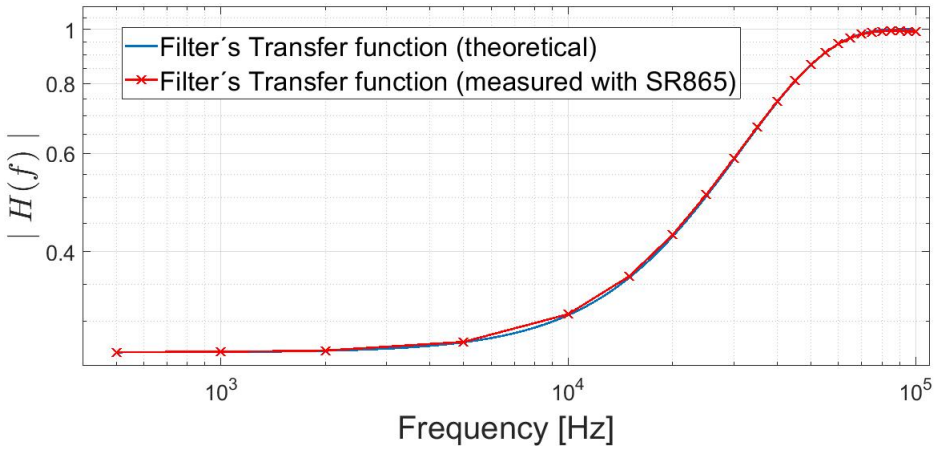

Fig. 11. High-pass filter's transference.

[7] S. Lahti, P. Sjövall, J. Vanne and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 5, pp. 898-911, May 2019, doi: 10.1109/TCAD.2018.2834439.

[8] M. J. Oliva. "Signal processing in FPGA". [Online] https://github.com/ushikawa93/signal_processing_fpga (accessed on November 2022).

[9] Intel Corporation. "Cyclone V Device Overview". [Online] https://www.intel.com/content/www/us/en/docs/programmable/683694/current/cyclone-v-device-overview.html (accessed on November 2022).

[10] Terasic Technologies Inc. "DE1-SoC User Manual". [Online] www.terasic.com.tw/ (accessed on November 2022).

[11] Intel Corporation. "Nios® II Software Developer's Handbook". [Online] https://www.intel.com/content/www/us/en/docs/programmable/683525/21-3/software-developer-s-handbook-revision.html (accessed on November 2022).

[12] Intel Corporation. "Altera IP Core user guide". [Online] https://www.intel.com/content/www/us/en/docs/programmable/683359/17-0/altera-phase-locked-loop-ip-core-user-guide.html (accessed on November 2022)

[13] Intel Corporation. "Implementing Fractional PLL Reconfiguration with Altera PLL and Altera PLL Reconfig IP Cores". [Online] https://www.intel.com/content/www/us/en/docs/programmable/683640/current/implementing-fractional-pll-reconfiguration-33682.html (accessed on November 2022).

[14] Intel Corporation. "Intel FPGA Avalon FIFO Memory Core". [Online] https://www.intel.com/content/www/us/en/docs/programmable/683130/21-4/intel-fpga-avalon-fifo-memory-core.html (accessed on November 2022).

[15] Intel Corporation. "Avalon® Interface Specifications". [Online] https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html (accessed on November 2022).

[16] Hathwalia, Shruti, and Meenakshi Yadav. "Design and analysis of a 32 bit linear feedback shift register using VHDL". Indian Journal of Pure and Applied Physics (IJPAP), 2015, vol. 52, no 3, p. 203-209.

[17] "GNU Scientific Library: Other random number generators". [Online] https://www.gnu.org/software/gsl/doc/html/rng.html#other-random-number-generators (accessed on November 2022).

[18] Linear Technology. "LTC2308 - Low Noise, 500ksps, 8-Channel, 12-Bit ADC". [Online] https://www.analog.com/media/en/technical-documentation/data-sheets/2308fc.pdf (accessed on November 2022).

[19] Analog Devices. "AD9248 (Rev. B)". [Online] https://www.analog.com/media/en/technical-documentation/data-sheets/AD9248.pdf (accessed on November 2022).

[20] Terasic Technologies Inc. "THDB-ADA High-Speed A/D and D/A Development Kit User Manual". [Online] https://www.terasic.com.tw/ (accessed November 2022).

[21] Analog Devices. "AD9763/AD9765/AD9767 (Rev. G)". [Online] https://www.analog.com/media/en/technical-documentation/data-sheets/AD9763_9765_9767.pdf (accessed on November 2022).

[22] Mono Project. [Online] https://www.mono-project.com/ (accessed on November 2022).