Pre-Silicon FEC Decoding Verification on SoC FPGAs

Víctor Fernández, Carlos Abad, Ángel Álvarez, Íñigo Ugarte, Pablo Sánchez

***Abstract*—Forward error correction (FEC) decoding hardware modules are challenging to verify at pre-silicon stage, when they are usually described at register-transfer (RT)/logic level with a hardware description language (HDL). They tend to hide faults due to their inherent tendency to correct errors and the required simulations with a massive insertion of inputs are too slow. In this work, two verification techniques based on FPGA-prototyping are applied in order to complement the mentioned simulations: golden model vs implementation matching with thousands of random codewords and codeword/bit error rate (CER/BER) curve computation. For this purpose, a system on chip (SoC) field-programmable gate array (FPGA) is used, implementing in the programmable hardware part several replicas of the decoder (exploiting the parallel capabilities of hardware) and managing the verification by parallel programming the software part of the SoC (exploiting the presence of multiple processing cores). The presented approach allows a seamless integration with high-level models, does not need expensive testing/emulation platforms and obtains the results in a reasonable amount of time.**

***Index Terms*—Verification, Platform FPGAs, Prototyping, Emulation, BER/CER testing.**

I. Introduction

THE goal of forward error correction (FEC) components is to correct errors introduced in the channel, during transmission, in a communication system. Due to their correcting nature, FEC decoders are hostile to verify as they tend to hide errors. Typical values to be tested are the primary outputs, relevant internal points and performance parameters like the number of iterations (in iterative decoding algorithms) or the global bit or codeword error rate (BER/CER). For all of them, the use of a massive amount of inputs is required. At pre-silicon stage, when the decoder is typically described with VHDL or Verilog, all that information can be easily attained by simulation, but it is too slow.

In order to simulate FEC decoders and, particularly, when the BER/CER performance is computed, the typical simulation scheme comprises models for the coding (including random generation of inputs), modulation, channel (with a concrete noise model), demodulation and the decoding itself. Error rates have to be measured, for a range of E_b/N_0 values [1], up to very low values in some cases, which implies long Monte Carlo simulation times.

As an example, the low-density parity-check (LDPC) decoder detailed in the results section of this paper and used in Tele-Command (TC) space communications is, typically, analyzed up to CER=10^{-6} but lower values can be needed in other applications.

At algorithmic level, a software simulation with a high-level language, like C/C++ or Matlab, could be fast enough for many channel/coding/decoding combinations. When needed error rates are very low, or the channel models require the simulation of a big amount of input scenarios, even these execution/simulation strategies can be excessively slow.

In pre-silicon phases, field programmable gate array (FPGA) prototyping and emulation are the two main verification alternatives to hardware description language (HDL) simulation.

In FPGA prototyping [2], the module under verification and, usually, the simulation testbed, are integrated into an FPGA. By using the parallelization potential of programmable hardware, the verification procedure is highly accelerated. Some approaches use predefined library components to verify the performance of a communication system [3]. This is not useful for new developments which cannot be modeled with such previous components. Input generation and noise addition are relevant tasks as the used random values need to fulfill high quality distributions. Some approaches use simple, poor accuracy, hardware random generators [4]. Others must develop complex ad-hoc hardware implementations [5], [6].

Concerning emulation tools like [7], [8], they can provide a great degree of debugging capabilities for hardware/software (HW/SW) systems, but they are too slow for our Monte Carlo simulation goal.

In post-silicon stages, the typical verification metric for a FEC decoder is restricted to a BER/CER computation. The error rate measurement can be faced with commercial [9] bit error ratio testers. They are expensive but ideal for final or close to final products. There are also alternative arrangements, based on FPGAs for the generation of inputs and the testing of outputs [10].

Making the decoder testbed in hardware implies some challenging issues. It has to be a translation of the original high-level model and such a task cannot always be possible or leads to a very complex HW (generation of random numbers with a good Gaussian distribution is an example). Moreover, any change in specification implies a hardware re-design which is more time consuming than a software counterpart.

This paper shows a proposal to perform Monte Carlo simulations of pre-silicon FEC decoders by taking advantage of the capabilities of system on chip (SoC) FPGAs. The decoder is synthesized and integrated in the Programmable Logic (PL) part of a Xilinx SoC FPGA and the input generation and output checking are managed in the Processing System (PS) part by

Manuscript received XX; revised XX...This work has been supported by Project TEC2017-86722-C4-3-R, funded by Spanish MICINN/AEI.

The authors are with the Microelectronics Engineering Group, University of Cantabria, 39005 Santander, Spain (e-mail: victor,carlosab,alvarez,ugarte,sanchez@teisa.unican.es)

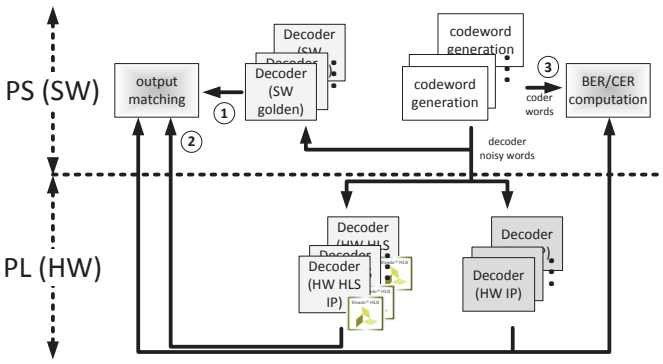

Fig. 1. Global view of the proposed verification flow.

software. By doing this, the verification does not need any other additional equipment nor any hardware translation of the decoder testbed allowing for a rapid setup.

The methodology keeps the matching with high-level simulations, allowing to compare the decoder under verification with a golden model reference. Moreover, by having the software and hardware parts in the same integrated device, the communication bandwidth is optimized.

In order to accelerate the computations, several instances of the decoder will be implemented in the PL and all the processors of the PS will be involved in a parallel execution by programming the software with OpenMP. With this, a BER/CER computation or a matching with a golden model by using a massive quantity of inputs is feasible in a practical amount of time.

Next section of the paper shows all the details of the proposed verification procedure. Section III reports the results of applying the methodology to a LDPC decoder using a Zedboard and a ZCU102 board. The conclusions are wrapped up in Section IV.

II. Verification Approach

The verification procedure proposed (Fig. 1) provides two types of analysis. First, the approach is applied to check if the decoder generates the same outputs as a high-level golden model for a set of thousands of random inputs. Values of the golden model can be obtained from a software function (labeled as 1 in Fig. 1) or from a HW intellectual property (IP) component generated by the designer from high level synthesis (HLS) of the software function (labeled as 2 in Fig. 1). This first approach is detailed in Section II.B. Second, the outputs of the decoder are compared with the ones generated by the coder in order to compute the CER/BER. This is labeled as 3 in Fig. 1 and detailed in Section II.C.

HW/SW Integration is made with Xilinx SDSoC tool [11]. It provides automatic methods to convert software functions to hardware (by HLS) and, also, to seamlessly integrate user predefined components (the HDL FEC decoder in our case) associating them to software functions. More details of the initial setup in the following section II.A.

A. Decoder Integration into the Verification System

The first step in the methodology is to create an Advanced eXtensible Interface (AXI) IP component from the decoder to be verified. The decoder has, apart from usual clk, $reset$ and other control pins, arrays of n elements for the input codeword and the output corrected word. If the decoding process is soft-decision, which is the most usual case in current decoders with high correction capability, the elements of these arrays are represented by a tuple of bits: m_1 bits for the input codeword and m_2 for the corrected output. As an example, the decoder used in the results section is a (128,64) LDPC decoder specified in [12] with 3 bits soft-decision at the input and hard-decision at the output. Therefore $n = 128$, $m_1 = 3$ and $m_2 = 1$.

Streaming protocol is added to the HDL code for n transfers at the input and the output per codeword. With that, Xilinx IP Integrator tool automatically generates an AXI-Stream IP (IP-XACT encapsulated) to be added to the IP library.

In SDSoC, a C-Callable library is defined and the previous HW decoder IP is added to it. The IP is associated to a function declaration with two arguments: *void decoder_hw(int in_r[n], int out_r[n])*. Let's suppose there is another function *decoder_sw* with the same arguments but with the C/C++ golden model definition of the decoder. The m_1, m_2 parameters could be defined by using a bitwise type instead of *int* type but we preferred to keep it in this way as it is more common in the original model. From this point, when the function *decoder_hw* is called in the user code, the decoding is, actually, performed on the programmable PL part of the SoC FPGA by the HW IP. The necessary HW/SW infrastructure (software drivers and communication HW) is also inferred. When *decoder_sw* is called, the routine is executed by the general-purpose CPU in the PS part of the SoC.

B. Golden Model Matching

The verification process is made by comparing the outputs of the HW IP decoder with the ones generated by a C/C++ golden model reference description (*decoder_sw* function). This can be done with HDL simulations, but the advantage of our approach is that thousands of random inputs can be applied in a reasonable amount of time. Usually, outputs to compare are the primary ones (decoded word for a decoder), but the extraction and comparison of other internal nodes can also be addressed.

Fig. 2 shows a Computation Unit that is managed by each processor of the PS. It is in charge of making the decoding of one codeword (one per decoder, actually) each time it is executed. The software part oversees the generation of the inputs for the decoders. It is executed by a software thread controlled by OpenMP as detailed below. The connection between the PS and the PL part is formed by a HP_PORT/DMA_DataMover combo which works optimally with AXI-Stream IPs. This communication infrastructure is automatically generated by SDSoC via designation with specific pragmas.

In order to increase the number of codewords decoded by unit of time, more decoders are added to the PL part to work in parallel. The number of possible decoders is limited by its

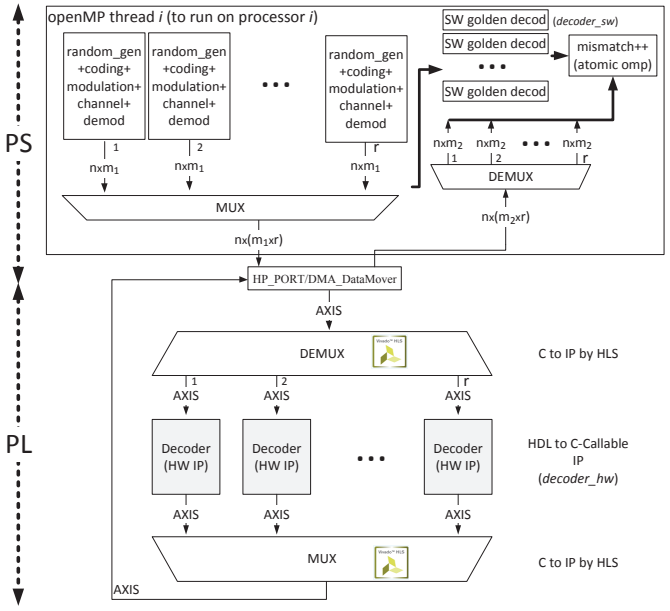

Fig. 2. Computation Unit.

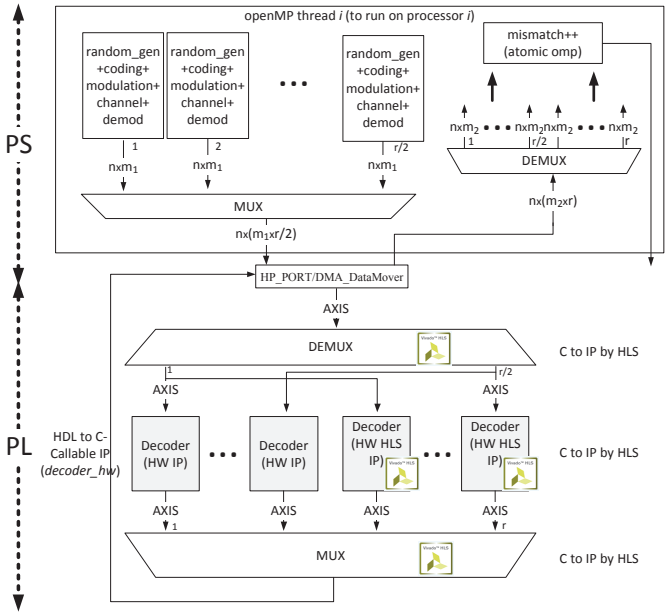

Fig. 3. Matching of the HW IP outputs with the ones generated by a HLS HW IP module synthesized from the C/C++ golden model.

size and the capacity of the PL. In the PS part, the generation section must provide a codeword for each decoder.

The number of High-Performance (HP) ports is limited in the Xilinx SoC FPGAs and in order to optimize the HW/SW communication speed, the inputs and outputs, typically a few bits wide, are grouped into a single word before they are transferred from HW to SW and vice versa. This is represented by the *mux* symbols in Fig. 2. In the PS part, the *mux* is, in fact, a simple function which works bitwise. On the PL part, the necessary *mux-demux* functionality can also be described as C/C++ functions and converted to HW automatically with SDSoC, which eventually calls the Xilinx Vivado HLS tool. This *mux-demux* communication procedure is limited by the number of bits of the inputs/outputs and the number of bits of the HP port (32/64 bits typically).

Concerning the control of the execution, C and the OpenMP programing interface are used with directives for the parallel execution management. At the start, it executes a sequential section where the random input generation is configured. There are two points where random values are needed: generation of information words and addition of noise. Input coder values (information words) are generated following random distribution defined by the SFMT method [13] which is a speed-up variety of the original Mersene-Twister 19937 random generator with a period of 2^{19937} – 1 values [14]. In addition, the noise model applied to the channel is the classic Additive White Gaussian Noise (AWGN). To include it, random normalized values are generated based on Ziggurat method, following [15]. Values from both random sources are consumed in parallel by several decoders. In order to avoid correlated sequences, seeds are generated by using the *shr3_seeded* function.

It is not the purpose of this paper to fix a way to generate random values but to provide a simple method to apply such values to the system. As it is managed in software, an alternative input generation procedure or noise model can be integrated without any change in the methodology.

After the initial phase, the execution follows with a double nested loop. The outer one iterates on the values of E_b/N_0. The inner one iterates on codewords. This inner loop is parallelized with the *#pragma omp for* directive. Inside the loop, the generation of codewords, the decoding and the output comparison are discriminated for each thread. The number of used threads is equal to the number of available processors in the SoC FPGA. As an example, 2 processors in a Zedboard platform and 4 processors in a ZCU102 board. There are p threads and each of them works with r decoders. Therefore, the number of codeword loop iterations is divided by p and r (which is the overall degree of parallelism).

In order to compare the outputs of the HW IP with the ones of the software model two options are considered: the first one is to run the C/C++ reference function on the PS part of the SoC FPGA and compare the output with the one generated by the HW IP. This approach can be seen in Fig. 2. r instances of the *decoder_sw* function are launched in order to compare with the r outputs of the HW IPs. If any output differs, the *mismatch* global variable is increased (protected with a *#pragma omp atomic* clause, as the variable is shared between threads). If the decoder works properly, such variable should remain at zero. The second option is to generate a HW IP from the C/C++ software function, by HLS, and compare its outputs with HW IP under verification. This can be seen in Fig. 3.

The generation of a HW IP from the original C/C++ reference model may be challenging. Not all C/C++ code is synthesizable, and some transformations could be necessary. The elimination of dynamic memory or the global variables are two examples. In addition, the circuit resultant from HLS is very dependent on synthesis pragmas.

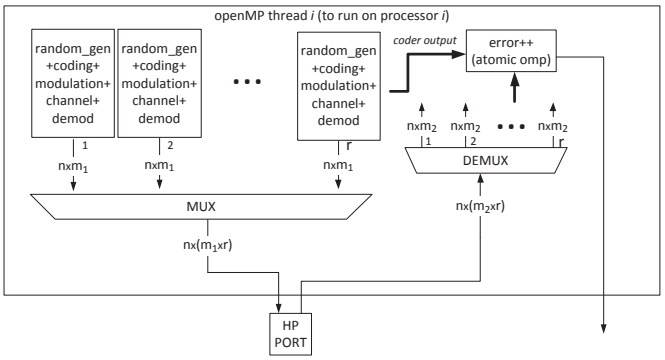

Fig. 4. PS part for CER/BER computation. PL as in Fig. 2.

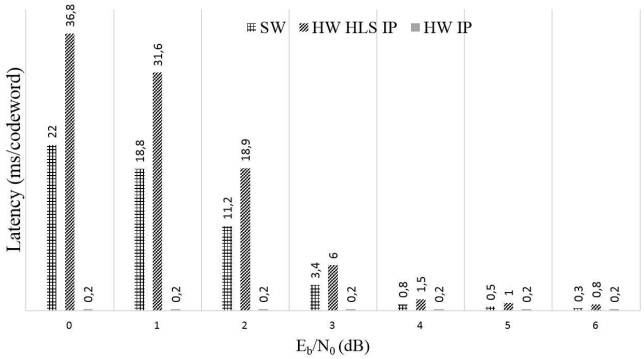

Fig. 5. Latency comparison for three implementations of the decoder.

As can be seen in Fig. 3, the PS part generates, for this matching option, $r/2$ codeword inputs, as they are injected to both types of HW IPs. In the PL part, there are $r/2$ HW IPs and $r/2$ HW HLS IPs. They generate r outputs in total that will be compared in the PS part.

C. BER/CER Computation

Another way to complement the verification process is by computing the CER/BER of the decoder. To do this, the configuration of the PS part can be seen in Fig. 4. The PL part is equal than the one shown in Fig. 2.

Regarding the control software, the inner loop iterates on codewords, injecting an amount of them that produces around 100 errors (classic value in the field which gives enough confidence in the value of error rate obtained) in the decoding results. Inside each thread, whenever the output (any bit for BER and the complete codeword for CER) of the decoder is different than the output of the coder, for a given codeword, the global variable *error* is incremented. When the output of the decoder is soft-decision, a simple conversion to hard-decision is performed before comparison. For each E_b/N_0 value, after the codeword loop, the BER/CER value is recorded.

III. Results

The proposed verification methods have been applied to a (128,64) binary LDPC decoder specified in [12] by the Consultative Committee for Space Data Systems (CCSDS) for Tele-Command (TC) space communications. The HDL description to be verified has been recently designed for a project by the European Space Agency (ESA) following a common partial parallel architecture [16]. The trade-off between resources and throughput was focused to fulfill the project requirements. The algorithm used for decoding was the Normalized Min Sum. The methodology can be applied to other types of decoders (Reed Solomon, BCH, Polar, etc.) in the same way.

Before showing verification results, it is opportune to characterize our decoder in the three versions that are going to be implemented: software golden model (next labeled as *SW* and named as generic *decoder_sw* in the methodology), HW IP obtained from the HDL description (labeled as *HW IP*), which is, in fact, what we want to verify and HW IP obtained with HLS from the software version (labeled as *HW HLS IP*).

The characterization was performed in a Xilinx ZCU102 platform, featuring a Zynq UltraScale MPSoC with 1.2 GHz 4-core ARM Cortex-A53 integrated with FPGA logic. In terms of used resources, both HW versions are not remarkably different. In terms of latency, results are shown in Fig. 5.

To get latency data, a basic arrangement made up of input generation, one decoder and output checking was defined. For the software version, no decoder was implemented in the PL part. For the other two cases, the decoder was implemented in the PL part. The latency was obtained by getting execution time before and after the decoding execution and computing the difference. For HW versions, the latency measured includes the delay of HW/SW communication.

Latency data of Fig. 5 are represented for the values of E_b/N_0 used in this section, that is, from 0 dB to 6 dB. As this parameter grows, the latency decreases. This is because the LDPC decoding algorithm is iterative and it needs less iterations to get a solution as the signal to noise ratio increases. The HW IP manually designed at RT level outperforms by far the others. The HW IP obtained by HLS is much slower. It is even slower than the pure SW decoder.

When applying the approach presented in Section II.B, results can be seen in Table I. 10^5 codewords per E_b/N_0 (from 0 dB to 6 dB) were injected. No mismatches were detected between the HW IP and the references. For the used decoder, the matching with the HW IP obtained with HLS is slower. This is because this HW HLS IP introduces a very slow path as it was anticipated with latency results shown in Fig. 5.

The speed of the HLS IP could be improved by struggling with HLS pragmas, but that task is out of the scope of the presented approach, which is limited by providing a way to use the IP generated by HLS.

The application of the method explained in Section II.C to the (128,64) CCSDS LDPC decoder was used to get a CER to E_b/N_0 curve which is the usual plot in this field (rather than BER). The execution times needed to get the results are reported in Table II. In the ZCU102 platform 16 replicas of the HW IP were used in parallel to get the complete plot in around 1 hour. To get the same plot by HDL simulation of the RT level description, around 80 days are necessary (obtained by extrapolating the time needed for a fraction of the inputs).

Fig. 6 shows the obtained CER curve. The values acquired

TABLE I
GOLDEN MODEL MATCHING FOR 10^5 MATCHES AT EACH E_b/N_0

Platform	Nº Thr.	Nº HW IPs per Thread	Nº soft-ware de-coders	Nº HW HLS IPs per Thread	Exec. time
ZCU102 (Zynq Ultrascale+ XCZU9EG MPSoC)	4	4	4	0	20 min
		2	0	2	22 min
Zedboard (Zynq-7000 XC7Z020 SoC)	2	4	4	0	93 min
		2	0	2	102 min
PC (Intel i7 @2.6 GHz. Xilinx Vivado HDL Simulator)	-	-	-	-	≈ 7 days

TABLE II
CER COMPUTATION RESULTS FOR E_b/N_0 IN [0,6] DB WITH 1 DB STEP

Platform	Nº Threads	Nº HW IPs per Thread	Exec. time
ZCU102	4	4	67 min
Zedboard	2	4	229 min
PC - HDL Simulation	-	-	≈ 80 days

match perfectly with those obtained by the golden C/C++ software model. This is because the software model used had all the hardware details, like quantization or saturation policies. If the software model is not so refined, a similar curve has to be obtained. In the same way, such kind of model could not be considered as a precise golden model and, therefore, it cannot be used in the approach described in Section II.B.

IV. CONCLUSION

This paper presents an approach for verifying pre-silicon FEC decoders by using SoC FPGAs. These devices are the only ones needed to perform the verification, keeping cost moderate and simple to set up. The PS part of the SoC is in charge of generating the massive and well distributed random inputs. Being software, the same models used in the high-level C/C++/Matlab simulations can be replicated. In addition, these high-level models can be used to match the outputs of the HW IP module under verification in two ways: by directly comparing software and hardware outputs and by comparing the hardware under verification with a hardware obtained by HLS of the software function which models the decoder. Another complementary way to verify the HW module is via generation of a BER/CER plot of the decoder. SoC resources are used extensively: ARM cores all work in parallel, and several copies of the HW IP are integrated in the PL part and perform codeword decoding in parallel as well. Results show that the time needed for verification is in the order of minutes, with the time required for HDL simulation in the order of days.

As a future work, we expect to apply the methodology to bigger codes. The use of HW memory buffers for execution in batches will be assessed to reduce idle periods during HW/SW transfers.

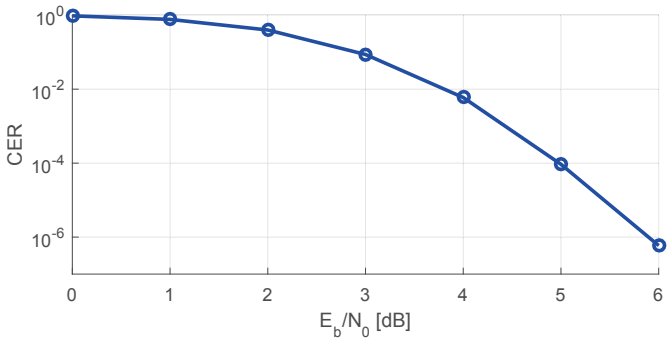

Fig. 6. CER curve obtained by applying the proposed approach.

REFERENCES

[1] M. C. Jeruchim, P. Balaban and K. S. Shanmugan, *Simulation of Communication Systems: Modeling Methodology and Techniques*, Boston, MA, USA:Kluwer, 2000.
[2] S. Sun and Z. Zhang, "Design of high-performance error-correcting codes using FPGA," in *Reconfigura Lbleogic: Architecture, Tools and Applications*, P.-E. Gaillardon, Ed. Boca Raton, FL: CRC Press, 2016, pp. 351-374.
[3] V. Singh, A. Root, E. Hemphill, N. Shirazi and J. Hwang, "Accelerating bit error rate testing using a system level design tool," *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, Napa, CA, USA, 2003, pp. 62-68.
[4] B. Unal, M. S. Hassan, J. Mack, N. Kumbhare and A. Akoglu, "Design of High Throughput FPGA-Based Testbed for Accelerating Error Characterization of LDPC Codes," *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, 2019, pp. 1-8.
[5] A. Alimohammad and S. F. Fard, "FPGA-based bit error rate performance measurement of wireless systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 7, Jul. 2014, pp. 1583-1592.
[6] F. Angarita, V. Torres, A. Pérez-Pascual and J. Valls, "High-throughput FPGA-based emulator for structured LDPC codes," *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*, Seville, 2012, pp. 404-407.
[7] Palladium Emulation Platform, Cadence Corporation. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-z1.html
[8] Xilinx Quick Emulator, Xilinx Corporation. Available: https://www.xilinx.com/support/documentation/sw_manuals /xilinx2017_2/ug1169-xilinx-qemu.pdf
[9] M8000 Bit error ratio tester. Keysight Corporation. Available: https://www.keysight.com/es/en/products/bit-error-ratio-testers.html
[10] Q. Li, F. Gong, G. Li and P. Song, "A Versatile FPGA-Based Multi-Rate and Multi-Channel Transmission Loss Tester," *2018 24th Asia-Pacific Conference on Communications (APCC)*, Ningbo, China, 2018, pp. 131-136.
[11] Xilinx SDSoC, Xilinx Corporation. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html
[12] CCSDS 231.0-B-3, "TC Synchronization and Channel Coding", Blue Book, Sept. 2017.
[13] M. Saito and M. Matsumoto, "SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods*, Springer, 2006, pp. 607-622.
[14] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, 1998, pp. 3-30.
[15] W. Tsang and G. Marsaglia, "The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, vol. 5, no. 8, Oct. 2000.
[16] F. Demangel, N. Fau, N. Drabik, F. Charot and C. Wolinski, "A generic architecture of CCSDS Low Density Parity Check decoder for near-earth applications," *2009 Design, Automation & Test in Europe Conference & Exhibition*, Nice, 2009, pp. 1242-1245.