SPL 2023
Welcome to SPL2023!
XI SOUTHERN PROGRAMMABLE LOGIC CONFERENCE
March 27th to 31th, 2023.
San Luis - Argentina

PROCEEDINGS

San Luis - Argentina
March, 2023

EditorsJuan Pablo Soto BarreraAlejandro

Welcome

The SPL2023 Organizing Committee, gives you welcome to the **XI Southern Programmable Logic Conference** in San Luis, Argentina.

The conference is organized by Universidad Nacional de San Luis, Argentina, and we aim to provide a high-level international forum for researchers and engineers to discuss recent advances, new techniques and applications in the field of reconfigurable logic technology. The XI edition continues the tradition of the previous one to become the meeting point for the worldwide community in the area.

SPL2023 topics include Embedded Processors and IP Cores, System-on-Chip, Computer Arithmetic, Image Processing and Vision, FPGA Architectures for Specific Applications, Fault Tolerance, Test & Verification, High-Performance Computing, Design Methodologies and Tools, High-Level Abstraction, Reconfigurable Computing, and Hardware/software co-design.

We would like to make a special acknowledgement of the contribution of our distinguished keynote speakers, the session organizers, the reviewers and all the authors. Your participation, and the spirit in which you undertake it, makes SPL2023 more successful.

Finally, we sincerely wish you a pleasant stay and a fantastic memory of San Luis.

Sincerely,
Operating Committee

Technical sponsorship

Institutional sporsonsing

Operating committee

General Chair

Julio Dondo Gazzano - Universidad Naciona de San Luis (UNSL, Argentina)

Program Co-Chairs

Carlos Vaderrama Sakuyama - Université de Mons, Department of Electronics and Microelectronics (SEMI, Belgica)

Fernando Rincón Calle - Universidad de Castilla-La Mancha (UCLM, España)

Proceedings Co-Chairs

Elias Todorovich - Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN, Argentina)

Juan Pablo Soto Barrera - Universidad de Sonora (UNISON, Mexico)

Designer Forum Chair

Cristian Sisterna - Universidad Nacional de San Juan (UNSJ, Argentina)

Financial Co-Chairs

Ricardo Cayssials - Universidad Tecnologica Nacional Bahia Blanca (UTN BHI, Argentina)

Cristian Falco - Universidad Nacional de San Luis (UNSL, Argentina)

International Relationship Chair

Gustavo Suter - Universidad Autónoma de Madrid (UAM, Spain)

Local Chair

Carlos Federico Sosa Paez - Universidad Nacional de San Luis (UNSL, Argentina)

Local Committee

Diego Costa - Universidad Nacional de San Luis (UNSL, Argentina)

Roberto Kiessling - Universidad Nacional de San Luis (UNSL, Argentina)

Publicity Chairs

Rodrigo Alejandro Melo - indie Semiconductor (Argentina)

Ivana Trento - Universidad Nacional de San Luis (UNSL, Argentina)

XI Southern Conference on Programmable Logic
SPL2023

Table of Contents
Full Papers

Hardware Acceleration of a CNN-based Automatic Modulation Classifier

Sravanth Chebrolu *, Srinivas Boppu*, Linga Reddy Cenkeramaddi†
cs21@iitbbs.ac.in, srinivas@iitbbs.ac.in, linga.cenkeramaddi@uia.no
*School of Electrical Sciences, Indian Institute of Technology Bhubaneswar (IITBBS), India
†Department of Information and Communication Technology, University of Agder, Norway

***Abstract*—Automatic modulation classification (AMC) has found its place in numerous applications, ranging from cognitive radio and adaptive communication to electronic reconnaissance and spectrum interference detection. Several attempts have been made to develop a high-accuracy modulation classifier using machine learning based convolutional neural networks (CNNs). This paper considers one such model, which uses a fixed boundary range empirical wavelet transform and deep CNN, and accelerates the model on the ZCU104 FPGA board to achieve fast classification times. The proposed accelerator can achieve a maximum classification accuracy of 96% for $+8$ dB signal-to-noise ratio (SNR) radio signals. Compared to similar works, the accelerator performs reasonably well for low SNR ratios ($\leq +6$ dB). Furthermore, the model is implemented on an edge CPU device (Raspberry Pi), and our accelerator is $50\times$ faster than the CPU implementation. Our design achieves a reasonable throughput of $1.8K$ classifications/sec and a classification time of $550\,\mu s$ per sample.**

***Index Terms*—Modulation Classification, Hardware Acceleration, Deep Learning, Convolutional Neural Networks, Vitis AI**

I. Introduction

There have been significant advances in wireless communication technologies and their standards recently. Understanding the radio spectrum in an autonomous manner plays an important role in numerous applications, such as electronic warfare, threat analysis in military scenarios, dynamic spectrum access, and spectrum interference detection [1], [2]. For instance, automatically identifying the modulation types of received signals allow the receiver to demodulate the signal; thus, the development of an efficient algorithm for modulation identification, also called *Automatic modulation classifier (AMC)*, is the priority in many software-defined radio-based communications [3]. AMCs have been extensively studied in recent years [3], [4], and several Deep Learning (DL) based techniques have emerged with huge Convolutional Neural Network (CNN) layers that have shown remarkable accuracy for automatically classifying modulated radio signals [5]. While these networks offer good accuracy, CNN-based networks are intrinsically slow due to the high computational complexity of the convolution operation. Since achieving fast classification times is crucial in several wireless communications applications, it is challenging to implement CNN-based AMCs for practical purposes. The complexity of the convolution operation can be greatly reduced if it is performed in parallel. Therefore, hardware accelerators based on FPGAs perform much better than traditional CPUs for CNN inference due to their parallel processing capabilities [6], [7]. Furthermore, with the recent advancements in FPGA technologies, FPGAs also emerged as potential candidates for AI hardware acceleration. Extensive research in this field has led to many new technologies in this space and frameworks such as *Vitis AI* have emerged, which enable AI inference acceleration on Xilinx FPGA platforms.

Vitis AI supports deep learning frameworks like TensorFlow and offers a suite of tools and APIs to prune, quantize, optimize, and compile pre-trained models to achieve the highest AI inference performance on Xilinx FPGAs [8]. In this paper, we propose a CNN-based hardware accelerator for AMC. We use the Vitis AI Development Kit, which relies on the Deep learning Processing Unit (DPU) at its core to accelerate inference for CNN-based models. The main contributions of this paper are

- Training and deploying a CNN-based AMC on the ZCU104.
- A model that achieves high classification accuracy even for low SNRs ($\leq +6$ dB).
- Speed and accuracy comparison of the FPGA implementation against a CPU-based edge device (Raspberry Pi).

The rest of the paper is organized as follows. Section II describes the related work. The problem statement is given in Section III and data set generation and training details are discussed in Section IV. Section V and Section VI discuss the DPU and the Vitis AI development flow. Finally, the results of our hardware implementation and conclusions are discussed in Section VII and Section VIII, respectively.

II. Related Work

Several ML-based techniques have been proposed for AMCs using radio signals in the last few years. For instance, In [10], Tridgell et al. proposed a real-time implementation on ZCU111 using the *radioML* data set for SNRs $\geq +6$ dB and achieved an accuracy of nearly 80% in the best case scenario (+30 dB SNR), a throughput of 488K classifications/s, and a classification latency of 8 μs. Similarly, in [5], Kumar et al. have also proposed a real-time implementation on the ZCU111 RFSoC—achieving a 94.46% maximum accuracy on the 24-class RadioML data set at +30 dB SNR while delivering a high throughput of 527K classifications/s, and a classification latency of just 7.5 μs. In [11], the authors have implemented a classifier for two modulation types, BPSK and QPSK, and achieved a classification accuracy of 100% for both at 10 dB SNR with a fast classification time of $42\,ns$. The definition of time to classify differs across

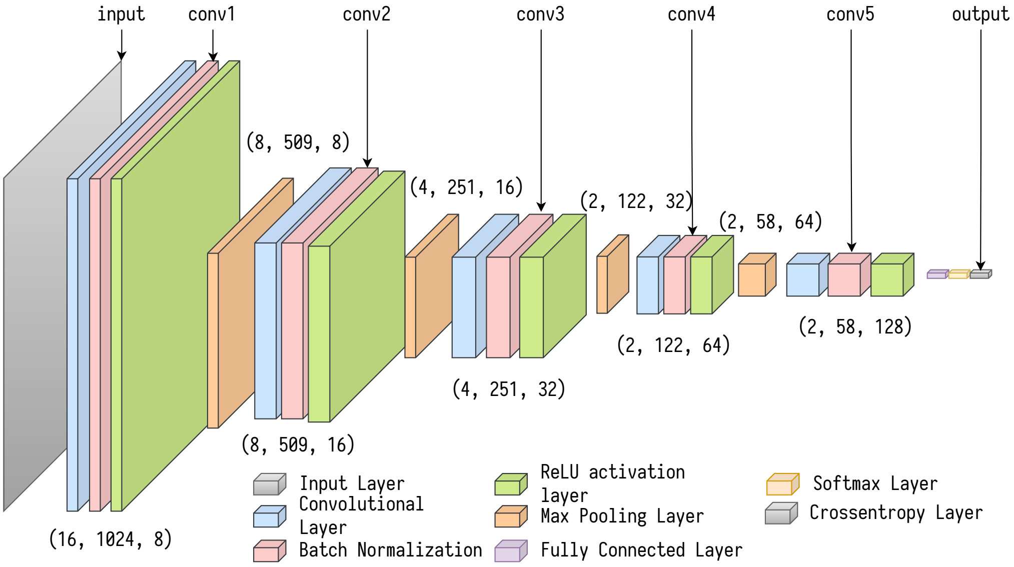


Fig. 1: Illustration of the CNN architecture used [9].

papers, with real-time implementations using latency to denote the time taken to classify the signal once the data is received. In [6], the authors achieved approximately 90% accuracy when the SNR was $\geq +4$ dB and the classification time was $20\,\mu$s for four modulation schemes BPSK, QPSK, 8-PSK, and 16-QAM. In [7], Soltani et al. have also worked on a real-time RF signal classification using Zynq UltraScale+ XCZU9EG FPGA and achieved a classification time of $24\,\mu$s, with an average accuracy of 94% over six modulation schemes; however, the SNRs used were not reported. In [12], Liu et al. also used a CNN model trained on the radio ML data set to achieve an accuracy of 72.5% for 0 dB SNR with a classification time of $4ms$. Our proposed hardware classifier implements the CNN architecture proposed by Yakkati et al. [9], with ReLU activations instead of tanh, in which a total of 9 modulation schemes [BPSK, QPSK, 64-QAM, PAM4, GFSK, CPFSK, B-FM, DSB-AM, SSB-AM] are used. The accelerator is implemented on the ZCU104 using Xilinx's *DPU* IP, a programmable engine dedicated to accelerating convolutional neural networks. Our classifier achieved the best classification accuracy of 96%, with a throughput of 1.8K classifications/s and a classification time of $550\,\mu$s. It is to be noted that the classification time includes the time delta to transfer the data from memory on the board through the on-chip ARM processor to the FPGA. The proposed work outshines previous implementations by a good margin even for low SNR radio signals, and achieves a modest classification speed.

III. Problem Statement

In the past, several AMCs were implemented using deep convolutional neural networks, and one such implementation which uses Fixed Boundary Range Empirical Wavelet Transform (FBREWT) and deep CNN is shown in Fig. 1. The CNN model comprises of five convolution layers, six batch-normalization layers, four max-pooling layers, one fully connected layer, as presented in Fig. 1. The input layer accepts pre-processed data which takes the shape of 16×1024 matrix, and the output layer is a 9×1 vector, which corresponds to the number of modulation schemes used for classification. Such deep neural networks can be deployed on edge computing devices like the Raspberry Pi. However, CNNs heavily depend on the convolution operation, which can be tedious to evaluate on pure CPUs due to their sequential nature of execution. When evaluated in parallel, this operation can be significantly speed up; therefore, FPGAs are among the best hardware platforms for implementing CNNs. Furthermore, modulation classifiers have not been implemented that fare well for low SNR radio signals. This work proposes a method to implement the CNN-based classifier that achieves high accuracy even for low SNR radio signals. The proposed method uses the Vitis AI tools provided by Xilinx to port the CNN model on Zynq FPGA to achieve hardware acceleration. Finally, we compare the performance metrics of the inference times with a raspberry pi 4B, which is an edge device with an ARM CPU.

IV. Data set, Preprocessing and Training

Table I summarizes the modulation schemes to be classified and the SNRs used in this work. A sizeable data set is necessary to attain good accuracy post-training. Therefore, for each signal-to-noise ratio, MATLAB was used to generate 9000 unique modulated signals (1000 signals per modulation scheme) by varying the AWGN channel levels between -4 dB to 10 dB.

TABLE I: Modulation types and signal-to-noise ratios used

Modulation Schemes	BPSK,QPSK, 64-QAM, PAM4, GFSK, CPFSK, B-FM, DSB-AM, SSB-AM
SNRs	10, 8, 6, 4, 2, 0, -2, -4

Then for each modulated signal, 16 sub-band signals (1024 samples in size) were generated by passing the modulated signal through the pre-processor discussed in section IV-A, which are then stacked together to form a 16 × 1024 image-like matrix, which the CNN receives as the input.

A. Empirical Wavelet Transform

Empirical Wavelet Transform (EWT) allows a multi-scale analysis of a time domain signal using an adaptive wavelet subdivision scheme. The EWT starts with segmenting the signal's spectrum and provides a perfect reconstruction of the input signal. The EWT coefficients partition the energy of the input signal into separate pass-bands [13].

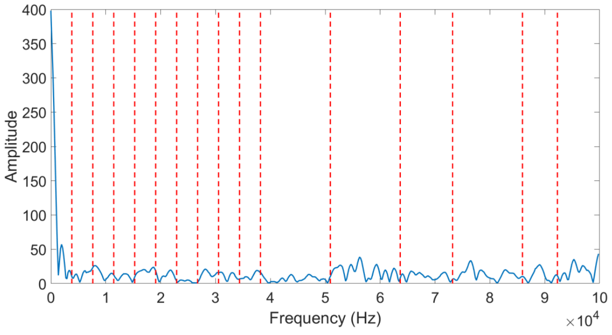


Fig. 2: Boundaries applied on the frequency spectrum [9].

In this work, we used Fixed Boundary Range EWT (FBREWT), where the frequency boundaries of the filter banks in the Fourier domain are pre-initialized, which means that regardless of the modulation scheme, the applied filter banks remain the same. Before creating the final data sets for training, the generated signals for each modulation scheme are pre-processed using FBREWT. We then perform the Fourier domain analysis on every modulated signal, extracting the sub-bands with the help of adaptive wavelet filter banks. For a given radio signal

$$r_n = [r(n)]_{n=0}^{N-1} \tag{1}$$

where N is the length of the radio signal, the Fourier domain signal is given by

$$R_k = \sum_{n=0}^{N-1} r_n \cdot \exp\left(j\frac{2\pi}{N}kn\right) \tag{2}$$

where k is the frequency at which the Fourier spectrum is evaluated, and it will be within the range $[0, f_s/2]$ where f_s is the sampling frequency of the signal. In the referred work [9], 16 pre-initialized filter banks are applied in the range $[0, \pi]$ segmented at $\mathbf{B}_l$ = [0.12, 0.24, 0.36, 0.48, 0.60, 0.72, 0.84, 0.96, 1.08, 1.20, 1.6, 2.0, 2.3, 2.7, 2.9], which corresponds to the following frequency segment array

$$\mathbf{F}_l = \frac{\mathbf{B}_l}{2\pi} f_s \tag{3}$$

Filter banks are applied for each of the 16 segments in $\mathbf{F}_l$, as shown in Fig. 2, to extract the sub-band signals using the *Inverse Discrete Fourier Transform (IDFT)* and trim each sub-band signal to 1024 samples. Thus, a 16 × 1024 sub-band matrix is generated after pre-processing. The final matrix for each modulation scheme for a given SNR is appended together to form a 9000 × 16 × 1024 array and then saved it in Matlab as a *.mat* file, which can be later imported to TensorFlow.

B. Training

The models were implemented for each SNR ratio in TensorFlow and the pre-processed data sets saved from MATLAB were imported in Python and fed to the CNN. Keras is used for network description and training and testing stages. Sklearn library was used to split the data set containing 9000 images into the training and testing data sets using the train_test_split function. This function takes x, y, and $test\_size$ as inputs. It returns x_train, y_train, x_test, and y_test as outputs, where x is the input image data array of 9000 images with each image of size $16 \times 1024 \times 1$, and y is an output labels array representing the modulation class corresponding to each element in x. The data set is split in a 9:1 ratio with the $test\_size$ chosen to be 0.10; therefore, the x_train and x_test contain 8100 and 900 images, respectively. Similarly, y_train and y_test contain 8100 and 900 labels, respectively. Training was performed for 100 epochs with a batch size of 128 per iteration using Google Colab, and each model took approximately 20 minutes to reach saturation in the validation accuracy.

V. Deep Learning Processing Unit

Xilinx's DPU is a programmable engine optimized for convolutional neural networks. The unit includes a high-performance scheduler module, a hybrid computing array module, an instruction fetch unit module, and a global memory pool module. The DPU uses a specialized instruction set, which efficiently implements many convolutional neural networks. Some convolutional neural networks deployed include VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, and FPN, among others. The DPU IP can be implemented in the *Programmable Logic (PL)* of the selected Zynq–7000 SoC or Zynq UltraScale+ MPSoC device with direct connections to the *Processing System (PS)*. The DPU requires instructions to implement a neural network and accessible memory locations for input and temporary and output data. A program running on the *Application Processing Unit (APU)* is also required to service interrupts and coordinate data transfers [14].

A. DPU Architecture

The internal architecture of the DPU consists of a scheduler module, processing engines (PEs), an instruction unit block, and a global memory pool module, see Fig. 3. The APU is the ARM processor on which the application will run, serves interrupts and data transfer from and to the DPU. The instruction unit handles reading and executing the instructions associated with the different operations of the accelerated CNN [16]. The *Fetcher*'s primary role is to fetch the DPU instructions associated with the model from memory. The decoder is responsible for decoding the instructions to drive the PEs. The dispatcher

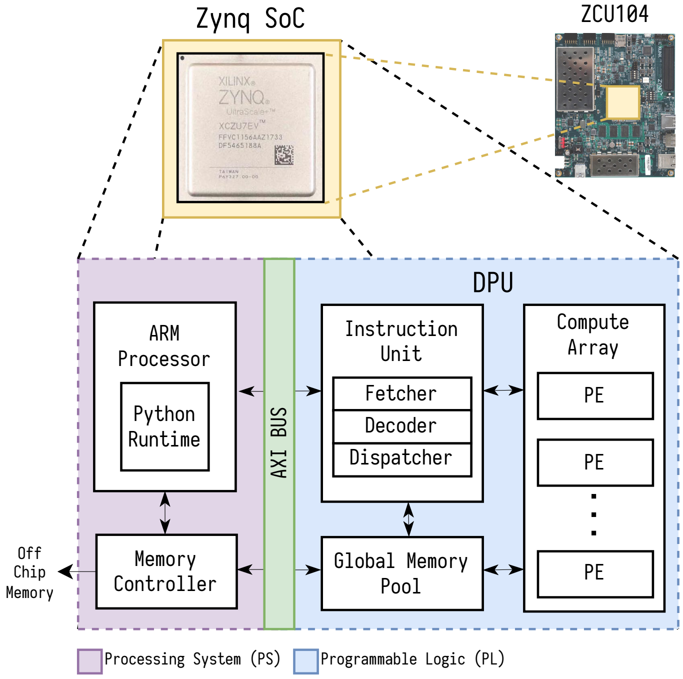


Fig. 3: ZCU104 Zynq UltraScale+ MPSoC evaluation board showing PS, PL, and internal architecture of DPU [15].

manages the data/instructions transfer among the PEs and the memory. The Global Memory Pool acts as a buffer for the input and output data and intermediate output from the DPU, which results in high throughput [16]. DPU is configurable and exposes several parameters which can be specified to optimize PL resources or customize enabled features. The DPU can be configured to meet the demands of a specific CNN architecture, which is why the DPU outshines other development flows. Xilinx's tools that are provided for configuring the DPU offer a lot of flexibility in choosing the framework (such as Tensorflow, Caffe and PyTorch), and the development flow uses the Vitis AI docker image, which contains all the necessary libraries to generate the instructions for the DPU.

VI. Vitis AI Framework

The Vitis AI development environment accelerates AI inference on Xilinx hardware platforms, including edge devices and Alveo accelerator cards. It consists of optimized IP cores, tools, libraries, models, and example designs. It is designed with high efficiency and ease of use in mind, unleashing the the full potential of AI acceleration on Xilinx FPGAs. It makes it easier for users to develop deep-learning inference applications by abstracting away the intricacies of the underlying FPGA.

The development flow is described in Fig. 4; the model is first trained in the TensorFlow framework using the data set generated from MATLAB. Training epochs are repeated until the validation loss and accuracy reach a saturation point, after which the model weights (which are in 32-bit floating point format) are exported to a file in *hdf5* format. The Vitis AI tools provide a model quantizer that supports all major frameworks, and it is used to convert the 32-bit floating point weights to 8-bit fixed point representation. This process uses a small calibration data set (in our case, it contains 900 samples) to minimize the accuracy loss due to a reduction in the model weight precision. After the quantization stage, the model is passed to the Vitis AI compiler, which converts the model graph to a set of domain specific instructions for the DPU unit; these instructions are saved to an *Xmodel* file which is loaded at run time. During inference, a Python script running on the APU acts as the mediator between the PS and PL; the script transfers the data from the on-chip memory to the DPU memory buffers. The DPU executes the instructions and the output classification vector is received in the Python run time. The details about the quantizer and compiler are briefly discussed in the next section.

A. Vitis Quantizer

In the context of DL, quantization is the process of representing the model weights in a smaller number of bits which reduces the numerical precision of the network, its complexity, and memory footprint, consequently resulting in reduced energy and storage costs [17]. Efficient quantization results in reduction in the overall model file size while mitigating the loss in accuracy. The Vitis AI quantizer relies on the *AdaQuant* [17] algorithm, which consumes a small calibration data set from training data without over-fitting and converts the numerical representation of the model weights from 32-bit floating point to 8-bit fixed point representation.

B. Vitis Compiler

The domain-specific compiler that comes with Vitis AI converts the quantized model into the appropriate sequence of instructions that drive the DPU. This is accomplished by identifying each layer and converting it into equivalent instructions. By the end of such a process, the main goal is to generate the kernels that shall be deployed on the FPGA and then used by API functions provided to drive the accelerators [16]. In this work, the Vitis AI compiler compiles the classification model after quantizing them into 8-bit fixed-point representation, which is the default post-quantization size.

C. Vitis API and Overlays

Overlays are hardware libraries that extend the user application from the PS into the PL. Overlays can accelerate a software application or customize a hardware platform for a particular application. For example, image processing is a typical application where the FPGAs can provide acceleration. We can use an overlay similar to a software library to run some of the image processing functions on the FPGA fabric. Similarly, the DPU can be considered an overlay that can be called from the PS using Vitis APIs to accelerate convolution layers on the PL. Initially, the data is stored in the off-chip memory, controlled by the DDR-controller through the AXI-Bus.

Model description in the off-chip memory is loaded onto the on-chip DPU buffers as a part of configuring the DPU for acceleration. Once the DPU finishes a process, the data is taken

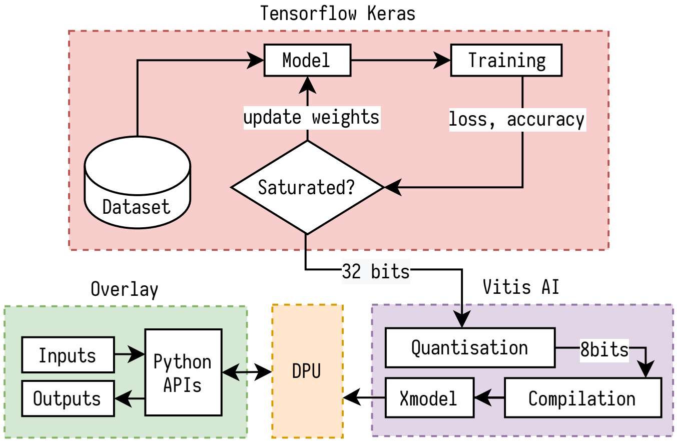


Fig. 4: Development flow to convert the model from TensorFlow to hardware implementation.

from the output on-chip buffers back to memory for any desired post-processing. The data includes the model input image, with a size of 16 × 1024, the model weights and biases, and model-associated instructions. The main objective of the Vitis API functions is to configure the DPU for the CNN model desired to be accelerated, which includes reading the DPU instruction sequence of the model and loading weights in the DPU [16]. Moreover, the API functions handle the data exchange between the CPU and the DPU, which allows the data to be pre-processed before being fed to the DPU or post-processed after carrying out the inference, which the DPU accelerates.

VII. Results

The CNN model was trained in TensorFlow to get similar accuracy as the MATLAB implementation described in the paper [9]. Table II depicts the accuracy comparison between the MATLAB and the TensorFlow implementations; we can observe that the accuracy is comparable; the 6 dB and 8 dB SNR take a $\leq 2\%$ dip in accuracy, whereas the rest are within $\pm 1\%$ range.

TABLE II: Comparison of obtained from the referred work with TensorFlow implementation

SNR(dB)	MATLAB(tanh) [9]	TF2(tanh)
10	96.44	95.91
8	96.89	95.49
6	93.56	90.34
4	89.11	87.21
2	82.22	81.93
0	73.11	74.44
–2	63.78	64.56
–4	59.56	61.87

The referred CNN model in [9] uses *tanh* activation functions. However, Xilinx's DPU does not support tanh activation function; therefore, the model had to be retrained using *ReLU* activation functions which posed quite a challenge since the learning rate at which the model reached saturation for tanh activation is different from ReLU. So a new learning rate scheduler had to be implemented that exponentially reduces the learning rate with the number of training epochs. Table III shows that the TensorFlow ReLU implementation is better than the TensorFlow tanh implementation and matches the MATLAB results from the referred paper. However, since the DPU implementation uses the quantized 8-bit weights, the model took a significant hit to the accuracy for ± 4 dB and -2 dB SNR with $\leq$ 5% dip in accuracy. Still, the other SNRs were within a 2% range. The average classification accuracy for the DPU implementation was 79.46% which is not that far from the 81.83% average accuracy in the referred paper.

TABLE III: Accuracy comparison for each SNR values across different implementations

SNR (dB)	Software Implementations			Hardware Implementation
	MATLAB(tanh) [9]	TF2(tanh)	TF2(ReLU)	DPU(ReLU)
10	96.44	95.91	97.22	94.44
8	96.89	95.49	96.67	96.00
6	93.56	90.34	93.56	92.78
4	89.11	87.21	89.11	85.44
2	82.22	81.93	80.89	79.69
0	73.11	74.44	73.56	69.33
–2	63.78	64.56	66.67	63.11
–4	59.56	61.87	60.00	54.89
Avg	81.83	81.46	82.21	79.46

A. DPU Performance and Resource Utilization

The DPU is connected to the ARM processor through an AXI bus on the FPGA chip to manage task scheduling and offloading weights and data to the DPU. AXI bus carries data and weights to the DPU. The DPU power consumption and resource utilization are optimized by leveraging special *UltraRAM* slices. The UltraRAM is a novel memory solution by Xilinx, which introduces higher memory speed with low energy consumption and resource utilization [18]. Table IV summarizes the resource utilization on the ZCU104 with the DPU, indicating that running the DPU on the fabric is quite a resource-intensive task.

B. FPGA vs CPU Performance

Perhaps the most exciting part of the current work is analyzing the performance against an edge device that is CPU only. For this purpose, a Raspberry Pi model 4 was chosen, which contains a Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz and 4 GB of RAM. The TensorFlow model was slimmed down using TensorFlow Lite and the inference metrics were extracted from both the CPU (with all four cores utilized) and the DPU implementations, which are summarized in Table V and Table VI, respectively. From the results, we can observe that the classification times per sample on the Raspberry Pi comes out to be around $24.5/900 = 0.0272$s, whereas the on the DPU it is approximately $0.5/900 = 0.00055$s. This roughly translates to a speed boost of $50\times$ the time taken by a quad core ARM CPU.

TABLE IV: DPU resource utilisation on the ZCU104 board.

Metric	LUT	LUT As Mem	Registers	BRAM	URAM	DSP Slices
Total Resources	227696	101516	456485	308	96	1728
Used by DPU	103171	11224	199093	290	92	1380
Resource Utilisation	45.31%	11.06%	43.61%	94.16%	95.83	79.86

TABLE V: Metrics on Raspberry Pi for validating 900 samples.

SNR(dB)	Throughput (FPS)	Time	Accuracy
10	36.32	24.777	97.22
8	37.54	23.971	96.67
6	37.51	23.988	93.44
4	36.97	24.344	89.11
2	37.05	24.286	80.89
0	36.24	24.830	73.56
–2	37.63	23.911	66.67
–4	36.67	24.542	60.00

TABLE VI: Metrics on the ZCU104 for validating 900 samples.

SNR(dB)	Throughput (FPS)	Time	Accuracy
10	1769.86	0.5085	94.44
8	1802.72	0.4992	96.00
6	1786.66	0.5037	92.78
4	1799.79	0.5001	85.44
2	1790.78	0.5026	79.69
0	1798.87	0.5003	69.33
–2	1765.70	0.5097	63.11
–4	1792.02	0.5022	54.89

VIII. Conclusions

In this paper, we proposed Xilinx's DPU-based hardware accelerator for an automatic modulation classifier based on FBREWT and deep CNN using the ZCU104 FPGA. A total of eight SNRs and nine modulation schemes were used (six digital modulation signals [BPSK, QPSK, 64-QAM, PAM4, GFSK, CPFSK] and three analog modulation signals [B-FM, DSB-AM, SSB-AM]). The details about data set generation in MATLAB and pre-processing were discussed briefly. The model weights were quantized from a 32-bit floating point representation to 8-bit fixed point representation with an accuracy loss margin of 5%. The model was compiled using the Vitis AI framework, which accepts TensorFlow model and generates instructions for configuring the DPU. Python APIs were used to transfer the data between the DPU and ARM chip on the Zynq SoC. The proposed accelerator could achieve an average classification accuracy of 79.46%, with the highest accuracy of 96.00%. It was successfully demonstrated that our FPGA hardware accelerator outperforms with respect to classification time when compared to the tflite model running on the Raspberry Pi containing the quad-core ARM CPU. The proposed accelerator was observed to perform reasonably well at low SNRs ($\leq +6$ dB) compared to similar works. The accelerator achieves a $50\times$ boost in classification speed compared to the CPU implementation using Raspberry Pi having a quad core Cortex-A72 (ARM v8) 64-bit SoC @1.5GHz. A throughput of $1.8K$ classifications/sec could be achieved with $550\,\mu s$ per classification.

References

[1] Z. Zhu and A. K. Nandi, *Automatic modulation classification: principles, algorithms and applications*. John Wiley & Sons, 2015.

[2] O. Dobre, A. Abdi, Y. Bar-Ness, and W. Su, "Survey of automatic modulation classification techniques: classical approaches and new trends," *IET Communications*, vol. 1, pp. 137–156(19), April 2007.

[3] T. Huynh-The, Q.-V. Pham, T.-V. Nguyen, T. T. Nguyen, R. Ruby, M. Zeng, and D.-S. Kim, "Automatic modulation classification: A deep architecture survey," *IEEE Access*, vol. 9, pp. 142 950–142 971, 2021.

[4] S. A. Ghunaim, Q. Nasir, and M. A. Talib, "Deep learning techniques for automatic modulation classification: A systematic literature review," in *2020 14th International Conference on Innovations in Information Technology (IIT)*, 2020, pp. 108–113.

[5] S. Kumar, R. Mahapatra, and A. Singh, "Automatic modulation recognition: An fpga implementation," *IEEE Communications Letters*, pp. 1–1, 2022.

[6] A. F. De Castro, R. S. R. Milléo, L. H. A. Lolis, and A. A. Mariano, "Artificial neural network based automatic modulation classification system applied to fpga," in *2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2021, pp. 1–6.

[7] S. Soltani, Y. E. Sagduyu, R. Hasan, K. Davaslioglu, H. Deng, and T. Erpek, "Real-time and embedded deep learning on fpga for rf signal classification," in *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019, pp. 1–6.

[8] V. Kathail, "Xilinx vitis unified software platform," in *Xilinx Vitis Unified Software Platform*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 173–174. [Online]. Available: https://doi.org/10.1145/3373087.3375887

[9] R. R. Yakkati, R. R. Yakkati, R. K. Tripathy, and L. R. Cenkeramaddi, "Radio frequency spectrum sensing by automatic modulation classification in cognitive radio system using multiscale deep cnn," *IEEE Sensors Journal*, vol. 22, no. 1, pp. 926–938, 2022.

[10] S. Tridgell, D. Boland, P. H. Leong, R. Kastner, A. Khodamoradi, and Siddhartha, "Real-time automatic modulation classification using rfsoc," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 82–89.

[11] M. Keshk and K. Asami, "Fpga-based automatic modulation recognition system for small satellite communication systems," 2017.

[12] X. LIU, J. SHANG, P. H. Leong, and C. LIU, "Modulation recognition using an fpga-based convolutional neural network," in *2019 22nd International Conference on Electrical Machines and Systems (ICEMS)*, 2019, pp. 1–6.

[13] J. Gilles, "Empirical wavelet transform," *IEEE Transactions on Signal Processing*, vol. 61, no. 16, pp. 3999–4010, 2013.

[14] A. Xilinx. (2019) Zynq dpu v3.1. AMD Xilinx. [Online]. Available: https://docs.xilinx.com/v/u/3.1-English/pg338-dpu

[15] J. Zhu, L. Wang, H. Liu, S. Tian, Q. Deng, and J. Li, "An efficient task assignment framework to accelerate dpu-based convolutional neural network inference on fpgas," *IEEE Access*, vol. 8, 2020.

[16] A. S. Hussein, A. Anwar, Y. Fahmy, H. Mostafa, K. N. Salama, and M. Kafafy, "Implementation of a dpu-based intelligent thermal imaging hardware accelerator on fpga," *Electronics*, vol. 11, no. 1, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/1/105

[17] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, "Improving post training neural quantization: Layer-wise calibration and integer programming," *CoRR*, vol. abs/2006.10518, pp. 2–3, 2020. [Online]. Available: https://arxiv.org/abs/2006.10518

[18] A. Xilinx. (2016) Ultraram: Breakthrough embedded memory integration on ultrascale + devices; technical report wp477. AMD Xilinx. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp477-ultraram

FPGA Implementation of Staggered Cellular Automata for Wave Propagation Simulation

Gustavo O. Pereira*, Santiago Guzman-Anaya†, Henrique G. Moura‡ and Daniel M. Muñoz*†
*Faculty of Gama, Electronics Engineering Undergraduate Program
‡Faculty of Gama, Automotive Engineering Undergraduate Program
†Department of Mechanical Engineering, Mechatronics Graduate Program, Automation and Control Group/GRACO
University of Brasilia, Brasilia, DF, Brazil
Email:gustavo.galvao@aluno.unb.br, santiago.anaya@aluno.unb.br, hgmoura@unb.br, damuz@unb.br

***Abstract*—The simulation of acoustic phenomena using a system resolution by ordinary PDEs is a work that demands a high computational cost. Using the phenomenon of propagation of acoustic waves in an ideal elastic medium in a steady state, it is possible to use a cellular automata system to simulate the phenomena. Even implementing a hardware solution using CA, the computational cost to simulate very large meshes is high. This work proposes a hardware architecture for a one-dimensional staggered cellular automata system model based on hybrid cells that can represent the basic wave phenomena. The proposed hybrid cell guarantees the simulation of different types of meshes without having to remodel the circuit. The proposed one-dimensional staggered cellular automata was able to successfully simulate 187 cells using a block composed of ten cells. The proposed solution enables the simulation of large meshes using FPGA devices with few resources.**

***Index Terms*—Cellular Automata, FPGA, Floating-point Arithmetics, Hardware-Software Co-design.**

I. Introduction

A partial differential equation (PDE) is a mathematical relation that may involve two or more partial derivatives of an undefined function. These kinds of equations are widely used in engineering modelings, because of their faithful representation of physical phenomenons [11]. However, the computational solution of PDEs is a hard numerical task and a time-consuming process [16].

Sound propagation is an engineering problem that can be stated in many physical situations and projects. Sound wave propagation can be described by the second law of motion of physical systems and, because of that, it is not easy to handle. Firstly, in practical simulations, the wave interacts with the involved complex geometries in space. Secondly, the wave equation that describes the sound wave propagation is a parabolic PDE, describing a time-dependent phenomenon [7].

In acoustic modelings, the PDEs need to consider many details on discrete signal processing, in addition to other application issues. Thus, working with PDEs may be useful for simple and small structure modelings, only.

Cellular automata (CA) modeling can be used to reduce the computational cost needed to solve PDEs. This numerical modeling approach considers a system of artificial cells that brings, inside their units, a very simple set of mathematical rules and well-defined states. In the case of sound propagation, in an ideal elastic medium and steady state, the well-known d'Alembert solution, applied to the one-dimensional wave movement, can be used to describe all the CA basic rules.

Recently, a hardware architecture of a CA was developed and implemented on FPGA devices, being able to emulate the basic wave phenomenons in one-dimensional systems [10]. The proposed organisms use three different cells to model dispersion, generation and hybrid mechanisms, and the results showed that the CA model fits the analytical solution. However, considering that each cell must represent a very small, and fixed piece of the studied environment, the authors stated that the solution may suffer with limited resources on the chip [9].

To overcome the above problem, this work presents a staggered solution for CA and proposes a hardware architecture for FPGA implementation. The staggered approach allows CA organisms to be divided into several pieces, called blocks of cells, that emulate wave propagation phenomena by using only the current state and the past state of each cell and its neighbors until the whole desirable calculation of the physical system is performed. The proposed approach can be implemented on small FPGA devices, in order to cover large physical systems.

In addition, this work proposes a generic cell for wave propagation in one dimension and its respective hardware implementation. The generic CA can be used to cover all the basic wave phenomenons, simplifying the hardware implementation of artificial organisms and improving the staggered approach efficiency.

Cellular automata and their implementation strategies have been recently studied for simulating different physical phenomena. An FPGA implementation of the popular CA system, called "the Conway's game of life" was implemented in [1] achieving an acceleration factor of 36,7 times than an equivalent GPU (*Graphic Processing Units*) application and, 2908 times faster than a traditional CPU solution.

The work presented by [14], brings an adaptive CA approach to the wave propagation problem, in two-directional space. The obtained results were compared to numeric and analytical models, available in the specific literature, with good convergence. A CA emulation of seismic events was implemented on a super-computer, obtaining results that exceed previous software simulated applications [5]. Another contribution was presented by [6] which developed a CA system to emulate structure-borne noise in one, two, and three-dimensional cases. The results pointed out a good convergence to the specific literature applied to the problem, which means that a new strategy becomes useful to the mentioned problem.

Jiménez-Morales *et al.* presented an alternative CA solution to the traditional PDEs model for laser's dynamic [4]. There were proposed variants to the traditional models, applied to different kinds of lasers. The obtained results were quantitatively validated for many real scenarios. The work presented by [2] brings a CA model able to emulate a ruptured biological diaphragm. The model was experimentally validated in a non-trivial problem.

Table I summarizes the related works using CA models. It can be seen, in the mentioned works, that only one is focused on the modeling of wave phenomenons propagation problems, considering its basic interactions with the obstacles in space. However, the obtained implementation holds some limitations related to the number of cells that could be parallelized in hardware, and, as a consequence, limits the size of the emulated physical system.

TABLE I
STATE OF ART

Authors	Year	Kind of solution	Plataforms
Bakhteri, Cheng, and Semmelhack [1]	2020	A SoC implementation of a CA system for the Conway's game of life.	FPGA
Shafiei, Khaji, and Eskandari-Ghadi [14]	2020	Adaptive CA system applied to sound wave propagation in an elastic lossless bidimensional case.	CPU
Lin and Zhao [5]	2020	Seismic events modeling using CA systems.	Super-computer
Luo, Wang, and Lei [6]	2021	A CA modeling applied to the emulation of structure-borne noises.	CPU
Jiménez-Morales, Guisado, and Guerra [4]	2018	Presentation of an alternative CA model applied to laser's dynamics.	CPU
Gupta, Gözen, and Taylor [2]	2019	CA system to emulate a ruptured biological diaphragm.	CPU
Moura and Muñoz [10]	2021	SoC implementation of a CA model to emulate acoustic wave phenomenons in one-dimensional space.	FPGA

II. BACKGROUD

A. Digitalized d'Alembert's Solution

Considering a plane wave propagating in the x direction, the propagation equation is [3]

$$\frac{\partial^2 p(x,t)}{\partial t^2} - \frac{1}{c_0^2}\frac{\partial^2 p(x,t)}{\partial t^2} = 0 \tag{1}$$

Manipulating this equation, it is possible to describe a solution based on the addition of two concurrent wave plots at any point in space [8] as follows,

$$p = f(x - c_0 t) + g(x + c_0 t) \tag{2}$$

This equation is known as d'Alembert's solution to the one-dimensional acoustic wave equation. Figure 1 represents the digitization of d'Alembert's solution for one-dimensional wave propagation. Notice that the sound pressure $p(n)$ is the sum of the plots p^+(n) and p^-(n), where n is the instant of time .

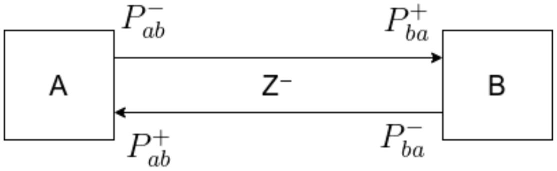


Fig. 1. Digitized d'Alembert solution for the one-dimensional wave. The output signal of point A is the input signal of point B, and vice versa [9].

Considering a generalized case where the waves can come from N different directions, the sound pressure at point J is calculated by

$$p_J = \sum_{i=1}^{2N} p_i^+ \psi_i, \tag{3}$$

where p_i^+ is the input sound pressure coming from direction i and ψ_i is the transmission coefficient. This equation is the mathematical model that will be used for the development of the CA model, as it represents the digitized d'Alembert solution for the one-dimensional wave. Figure 2 shows how the wave transmission and reflection phenomena occur within a medium of impedance Z_1 with an obstacle of impedance Z_2. In the figure, p^+ represents the portion incident on the obstacle, p^- as the portion reflected by the obstacle, and p^{tr} as the portion transmitted through the obstacle.

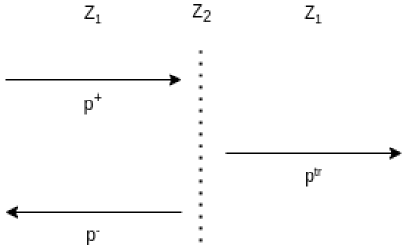


Fig. 2. Transmission and reflection of plane wave in a one-dimensional space. When passing through an obstacle, a part of the signal p^+ is reflected, and the other part p^{tr} is transmitted.

Considering that the characteristic impedance (R) of the medium can be determined by the ratio between sound pressure and the particles velocity, and assuming that the transmitted energy and the reflected energy are equal to the incident energy, the transmission coefficient can be stated as,

$$\psi_i = \frac{2R_i}{R_J + \sum_{i=1}^{2N} R_i} \tag{4}$$

Combining the equations 4 and 3, the solution to the one-dimensional wave transmission problem is obtained as,

$$p_J = \frac{2R_1}{R_1 + R_2} p^{+}_{right} + \frac{2R_1}{R_1 + R_2} p^{+}_{left}, \tag{5}$$

where R_J is the characteristic impedance in the joint J, R_i is the characteristic impedance of the front (point A) and rear (point B) joints, and p^{+}_{right} and p^{+}_{left} are the pressure on the right and left, respectively.

B. Camphs1D *System*

The cellular automata system *Camphs1D* (*Cellular Automata Modeling of One-dimensional Physical Systems*) was proposed and implemented in [10]. This model implements the phenomena of one-dimensional wave propagation and reflection using a CA system.

To develop this system, three types of cells were modeled considering the wave physics phenomena: a) a dispersion cell that transmits the signal that comes from its neighboring cells (eq. 6); b) a generation cell that generates a signal and transmits it to its neighboring cells (eq. 7); and c) a hybrid cell that receives a signal and transmits and reflects part of this signal (eq. 8). The symbols “+” and “-” represent propagation direction, α_r and α_t are the reflection and transmission coefficients, n is the time constant, and p is the pressure.

$$\begin{aligned} \left(p_i^{+}\right) &= \alpha_t * \left(p_{i-1}^{+}\right)_{n-1} \\ \left(p_i^{-}\right) &= \alpha_t * \left(p_{i-1}^{-}\right)_{n-1} \end{aligned} \tag{6}$$

$$\begin{aligned} \left(p_j^{+}\right)_n &= \left(p_f^{+}\right)_n \\ \left(p_j^{-}\right)_n &= \left(p_f^{-}\right)_n \end{aligned} \tag{7}$$

$$\begin{aligned} \left(p^{+}\right)_n &= \alpha_t * \left(p_{i-1}^{+}\right)_{n-1} + \alpha_r * \left(p_{i+1}^{-}\right)_{n-1} \\ \left(p^{-}\right)_n &= \alpha_t * \left(p_{i-1}^{-}\right)_{n-1} + \alpha_r * \left(p_{i+1}^{+}\right)_{n-1} \end{aligned} \tag{8}$$

It is worth mentioning that, although in a strictly physical sense the term “dispersion” represents velocity and frequency changes, we maintained the terminology and formulation of the digital waveguides stated in [15], in which the term “dispersion” was coined for the propagation of acoustic waves.

III. Generic one-dimensional cell model

The proposed cells represent a new cell model for all the functions implemented in [10]. In order to facilitate the implementation of complex CA systems, this work presents a hybrid cell hardware architecture, named HGCA, that behaves like any of the three cells (see Fig. 3). Deriving an equation from the architecture is not straightforward because the architecture mixes the behavior of three cells by using muxes that enable the evaluation of the necessary conditions to select the required behavior (dispersion, generation, or hybrid).

The HGCA cell consists of 5 floating-point multipliers, 2 floating-point adders, 2 3x1 multiplexers, 2 4x1 multiplexers, 1 OR logic gate, and 6 buffers signal. It has 14 input signals and 4 output signals of different sizes. The floating-point arithmetic representation was used to provide a large dynamic range if compared to fixed-point, allowing very small and very large numbers to be represented with the same bit-width [13].

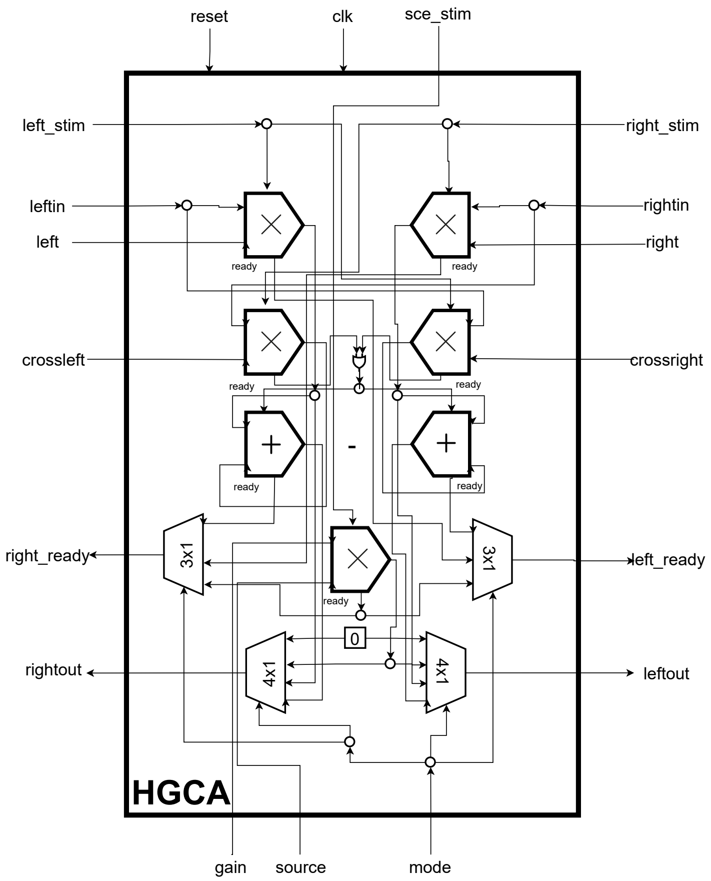


Fig. 3. Generic cell structure implemented in reconfigurable hardware. The HGCA cell is composed of 5 multipliers, 2 adders and 4 multiplexers.

Comparing the structure of the HGCA cell proposed in Fig. 3 with cells reported in [10], it is noted that the HGCA cell has more floating-point operations; however, it provides the flexibility of choosing which mechanism the cell will follow at each time instant.

Looking at the latency of each mode that the cell can operate, it can be noticed that a hybrid cell has the highest time consumption with 4 clock cycles. Both dispersion and generator cells, have a latency of 2 clock cycles, causing a delay in the system when a hybrid cell acts together with the others. Thus, some buffers were used to delay the *ready* signals in the faster modes. From this, the default latency time for this cell model is 4 clock cycles.

IV. Proposed Staggered Architecture

This architecture allows the cellular automata simulation to be divided into blocks of cells. Thus, small FPGA devices with fewer resources can simulate larger wave propagation systems.

Figure 4 depicts the proposed staggered cellular automata. The main idea is to have a separate block of cells communicating with a software application. The cell block represents the structure where the cells are stored and the application sends to each cell the mode of operation, and the neighbors' current and past states. The maximum number of cells that can be stored in the block depends on the logical resources assigned. The application block could be implemented in software with a processor or in hardware using a Finite State Machine.

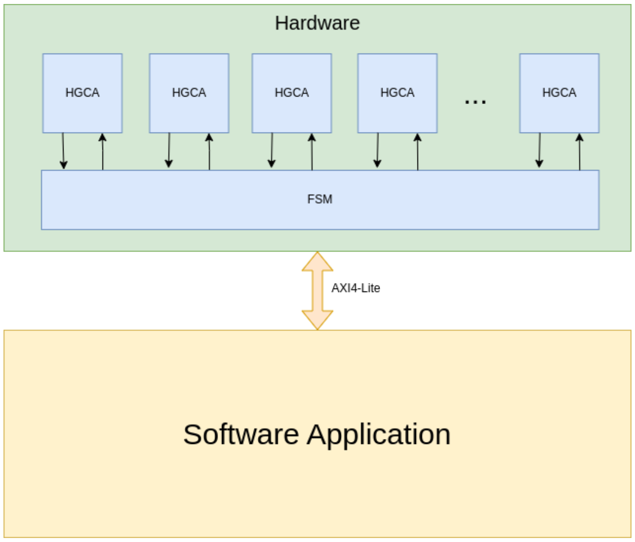


Fig. 4. Staggered architecture reference model with the generic cells. The communication is directly with the cell block.

In this work the staggered architecture was implemented in VHDL using floating point arithmetic operation IP-Cores, previously developed [13], [12]. Using the *FOR GENERATE* directive, it was possible to develop a parameterized solution for the number of cells the block can implement.

This cellular automata system emulates acoustics behaviors considering a homogeneous medium, i.e the same characteristics in all directions, and thin rigid walls that only produce reflections. Changes in the propagation medium were not considered in this work; however, a cellular automata system can implement this situation by adapting the distance between two neighboring cells (the higher the density, the closer the particles of the propagation medium). The distance between two consecutive cells can be implemented with registers that emulate the time delays representing the length between the cells. It is important to highlight that analytical methods, such as digital wave guides [15], commonly use the same sampling frequency and, as a consequence, only homogeneous mediums can be studied.

For comparison purposes, the same artificial organism *Camphs1DB9G2* proposed in [10] was used, allowing the response obtained by the staggered and non-staggered solutions to be compared. The organism Camphs1DB9G2 uses 187 cells, 3 hybrids, 2 generators, and the remainder are dispersion cells. The hybrid cells are at positions 2, 130, and 186. The first generator cell is at position 43 and operates for 16-time intervals, then changes to dispersion mode. The second generator cell is at position 93 and operates for 8-time intervals, then changes to dispersion mode. For the generator cells, two sinusoidal signals were created according to the sampling frequency of the system. In the staggered solution, the cell block was implemented with 10 parallel HGCA cells.

Testbenches with reading and writing capabilities were developed to test the proposed architecture. *Octave* scripts were developed to automatically get the text files from the generator cells.

For the physical implementation of the one-dimensional staggered cellular automata system model, two main components are needed: the cell block to calculate the outputs of each cell and a software application, which would be in charge of controlling the cell block inputs and modes of operations. In this work a hardware-software co-design was developed to integrate the cell block to the ARM processor of a Zynq 7020 System on Chip (SoC) device using the AXI4-Lite protocol. Thus, the information about the cells is controlled by the ARM processor. Using this principle, the mesh size limitation simulated by the FPGA will be associated with the memory size of the processor.

V. Results

A. Resources Occupation as a Function of the Bit-width

A numerical accuracy study with the proposed HGCA cell was done taking into account the bit-width of the exponent and mantissa words of the floating-point representation. The considered values for the mantissa word were 11, 13, 16 and 18 bits while the bit-width of the exponent word changed between 6 and 8 bits. For each proposed combination, the consumption of LUTs, FFs, DSPs and BRAM were collected after logic synthesis using Vivado and the Zynq 7020 SoC device from Xilinx. Table II shows the result obtained after the logic synthesis process for each of the cases.

TABLE II
Hardware occupation of the hybrid cell.

	Bits	LUT	FF	DSP	BRAM
EXP:8 FRAC:18	27	1125	314	5	0
EXP:8 FRAC:16	25	884	292	5	0
EXP:8 FRAC:13	22	740	259	5	0
EXP:8 FRAC:11	20	683	237	5	0
EXP:6 FRAC:18	25	1086	296	5	0
EXP:6 FRAC:16	23	819	274	5	0
EXP:6 FRAC:13	20	687	241	5	0
EXP:6 FRAC:11	18	617	219	5	0

A significant reduction in the consumption of LUTs and FFs was observed when the bit-width of the mantissa reduces from 18 to 16 bits. When decreasing the number of bits of the mantissa, the reduction still occurs, but in a smaller size. The reduction in the exponent size did not generate a significant change in the consumption of LUTs but had a slight reduction in the consumption of FFs.

For the sake of numerical comparisons with previous works, we decided to implement the proposed HGCA cell and the staggered CA architecture using a 27-bit floating-point representation (8 bits for the exponent word and 18 bits for the mantissa word).

B. Resource Occupation of the HGCA Cell

The generic HGCA cell proposed in Fig. 3 was encapsulated with an AXI4-Lite interface, to validate its physical implementation on a Xilinx Zynq 7020 FPGA SoC with a clock frequency of 100 MHz. Table III shows the resource utilization of the generic cell, with a consumption of 1119 LUTs, 303 FFs and 5 DSPs.

TABLE III
RESOURCE UTILIZATION OF THE HGCA CELL

Resource	Estimated	Available	Utilization%
LUT	1119	53200	2,10
FF	303	106400	0,28
DSP	5	220	2,27
BRAM	0	140	0

Figure 5 shows the circuit layout after the Place and Route process. In red is shown the occupation of slices by the HGCA cell. In blue, are shown the AXI4-Lite interface and other modules. One can conclude that it is possible to scale the number of cells in the Zynq 7020 device, as will be presented in the following subsection.

C. Characterization of the Staggered Cellular Automata

The validation of the proposed staggered CA architecture was carried out through behavioral simulations and from the physical implementation of a block of 10 HGCA cells using a Zynq 7020 SoC device at a clock frequency of 100 MHz.

Initially, the VHDL description of the artificial organism Camphs1DB9G2 (consisting of 187 cells) was validated in behavioral simulation using TXT files with memory values of each cells of the block at different positions and time instants.

Figure 6 shows the initialization of the simulation. Notice that the *firststart* signal initializes the simulation and, from here, the system starts reading new inputs into the cells when the cell block finishes calculating the outputs of the simulated cells. Over the course of the simulation, it can be noticed that the number of cells simulated at each time instant is increasing 10 by 10 with the *node_cnt* signal (in magenta). When simulating 187 cells at each time instant, the iteration counter is incremented (*it_cnt*). The simulation continues until 160-time iterations are completed.

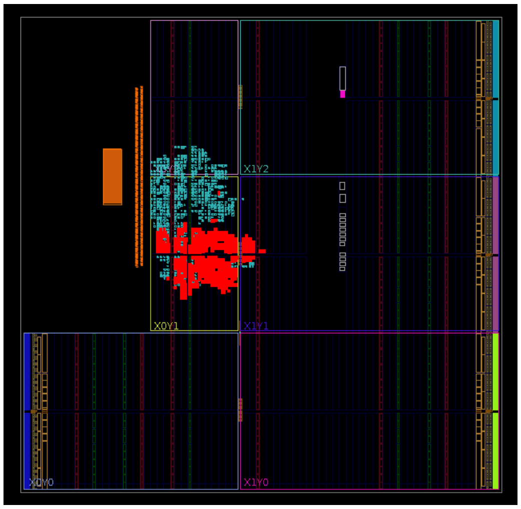


Fig. 5. The result obtained from PAR (*Place and Route*) after implementing the cell together with the ARM processor. In red is the region of logical blocks occupied by the HGCA cell. In cyan is the rest of the logical blocks occupied by the solution. In orange is the space occupied by the ARM processor.

During the simulation, the output values of a cell block were stored in a TXT file, which was used as input for the cell blocks in the next iteration. These files were also used to visualize and validate the output of the architecture from a script in *Octave*. For visualization purposes, the results of the outputs of each cell were summed left and right. With this result, a video was generated showing the outputs obtained from the proposed staggered architecture (https://youtu.be/BZG6xB7996U).

To simulate the required 160-time cycles of the 187 cells, 182.46 μs were required, thus achieving a throughput of 0.877 MOPS, whereas the non-staggered hardware architecture achieved a throughput of 25 MOPS [10]. On the other hand, the non-staggered solution, for the same system implemented in C (GCC compiler), required 0.7128 s running on the ARM processor and 0.1138 s for an AMD Ryzen7 3700U, 17.8GB RAM, Linux Mint 20.3.

Figure 7 shows the result of the staggered architecture mapped on a Zynq 7020 device. The proposed circuit was effectively mapped on the FPGA device using a clock frequency of 100 MHz. Table IV shows a resource utilization of the proposed architecture for the Camphs1DB9G2 system. A comparison with our previous work [10], which does not fit on the Zynq 7020 device, demonstrates that the proposed staggered dramatically reduces the resource occupation.

TABLE IV
RESOURCE UTILIZATION OF THE STAGGERED SOLUTION.

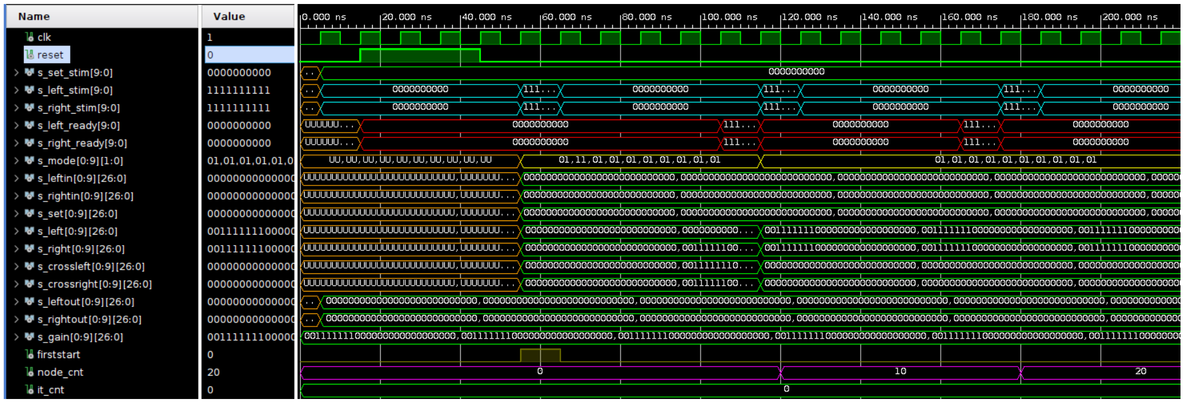
Architecture	Platform	LUTs	FFs	DSPs	BRAMs
Staggered (Total)	Z7020 Zedboard	12361	5877	50	2
Staggered (block cells)	Z7020 Zedboard	11052	3030	50	0
Non-staggered (block cells) [10]	ZU7EV ZCU104	22422	21924	372	0

The energy consumption of the staggered solution for a clock frequency of 100 MHz was approximately 0.142 W of static power and 1.565 W of dynamic power, being 96% of the dynamic power dissipated by the ARM processor.

VI. CONCLUSIONS

This work proposed a one-dimensional staggered cellular automata system to simulate acoustic wave propagation phenomena. The proposed solution is based on a generic cell structure that emulates the dispersion and reflection wave propagation mechanisms in a one-dimensional space. The model was able to simulate larger systems with fewer computational resources if compared to the previous solution in [10].

A hardware-software co-design was proposed for the integration of the staggered cellular automata system with an



Name	Value
clk	1
reset	0
s_set_stim[9:0]	0000000000
s_left_stim[9:0]	1111111111
s_right_stim[9:0]	1111111111
s_left_ready[9:0]	0000000000
s_right_ready[9:0]	0000000000
s_mode[0:9][1:0]	01,01,01,01,01,0
s_leftin[0:9][26:0]	00000000000000
s_rightin[0:9][26:0]	00000000000000
s_set[0:9][26:0]	00000000000000
s_left[0:9][26:0]	00111111100000
s_right[0:9][26:0]	00111111100000
s_crossleft[0:9][26:0]	00000000000000
s_crossright[0:9][26:0]	00000000000000
s_leftout[0:9][26:0]	00000000000000
s_rightout[0:9][26:0]	00000000000000
s_gain[0:9][26:0]	00111111100000
firststart	0
node_cnt	20
it_cnt	0

Fig. 6. Behavioral simulation result of the staggered solution with blocks of 10 cells for the Camphs1DB9G2 system. In cyan the left and right start signals, in red the ready signals, in yellow the mode signals, and in magenta the counter with the number of simulated cells.

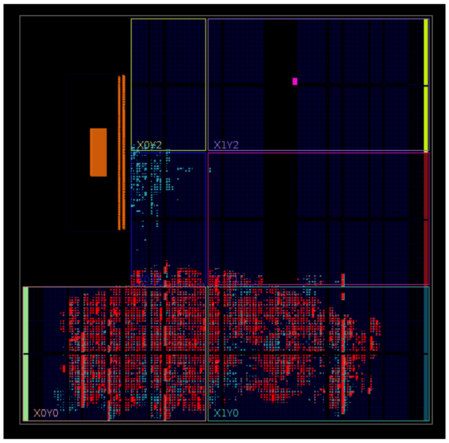


Fig. 7. Result obtained from PAR (*Place and Route*) after implementing the cell block with AXI4-Lite and the ARM processor. In red, there is the region of logical blocks occupied by the block of HGCA cells. In cyan, we have the rest of the logical blocks occupied by the solution. In orange, we have the space occupied by the ARM processor.

ARM processor. Thus, when integrating the processor with the DDR memory present in the development kit, the solution would be limited to the size of the DDR memory and not to the on-chip BRAM memory, which is always a scarce resource. This solution allows the simulation of thousands of cells, without being limited by the resources of the FPGA device.

As future works, we intend to replace the ARM processor with a Microblaze soft-processor, allowing a considerable reduction in energy consumption. It is also expected to develop a staggered cellular automata model in two dimensions, allowing real simulations of acoustic phenomena to be accelerated. Additionally, we intend to model wave phenomena using internal registers between the cells allowing different frequencies and propagation speeds to be emulated.

References

[1] Rabia Bakhteri, Julian Cheng, and Alex Semmelhack. Design and implementation of cellular automata on FPGA for hardware acceleration. *Procedia Computer Science*, 171:1999–2007, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[2] Abhay Gupta, Irep Gözen, and Michael Taylor. A cellular automaton for modeling non-trivial biomembrane ruptures. *Soft Matter*, 15:4178–4186, 2019.

[3] D.E. Hall. *Basic Acoustics*. Krieger, 1993.

[4] Francisco Jiménez-Morales, José Luis Guisado, and José Manuel Guerra. *Simulating Laser Dynamics with Cellular Automata*, pages 405–422. Springer International Publishing, Cham, 2018.

[5] Zhe Lin and Xiaohua Zhao. An improved approach to simulate seismic events based on cellular automata. *International Journal of Modern Physics C*, 31(04):2050053, 2020.

[6] Kun Luo, Zhenguo Wang, and Xiaoyan Lei. The cellular automata model of sound propagations and its application in structural noise calculations. *Applied Acoustics*, 182:108262, 2021.

[7] Friedrich Moser, Laurence J. Jacobs, and Jianmin Qu. Modeling elastic wave propagation in waveguides with the finite element method. *NDT & E International*, 32(4):225–234, 1999.

[8] H. Moura. Simulação da propagação de ondas acústicas através de uma malha de guias digitais de ondas., 2006. Disssertação (Mestrado em Engenharia Mecânica), UFU, Uberlândia, Brasil.

[9] H. Moura. Implementação em chip de sistemas celulares autômatos dedicados à emulação da propagação de ondas acústicas em sistemas físicos., 2022. Monografia (Bacharel em Engenharia Eletrônica), UnB, Brasilia, Brasil.

[10] Henrique G. Moura and Daniel M. Muñoz. Modeling wave propagation using cellular automata on chip. In *34th IEEE Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6, 2021.

[11] G.M. Murphy. *Ordinary Differential Equations and Their Solutions*. Van Nostrand, 1960.

[12] Daniel M Muñoz, Diego F. Sanchez, Carlos H. Llanos, and Mauricio Ayala-Rincón. FPGA based floating-point library for CORDIC algorithms. In *2010 VI Southern Programmable Logic Conference (SPL)*, pages 55–60, 2010.

[13] Daniel M. Muñoz, Diego F. Sanchez, Carlos H. Llanos, and Mauricio Ayala-Rincón. Tradeoff of FPGA design of a floating-point library for arithmetic operators. volume 5, pages 42–52, Journal Integrated Circuits and Systems, 2010.

[14] Masoud Shafiei, Naser Khaji, and Morteza Eskandari-Ghadi. An adaptive cellular automata approach with the use of radial basis functions for the simulation of elastic wave propagation. *Acta Mechanica*, 231(7):2723–2740, 2020.

[15] Julius O Smith. Physical modeling using digital waveguides. *Computer music journal*, 16(4):74–91, 1992.

[16] Holger Thies. Uniform computational complexity of ordinary differential equations with applications to dynamical systems and exact real arithmetic. *Graduate School of Arts and Sciences, University of Tokyo, Tokyo, Japan*, 2018.

Streamlining FPGA Circuit Design and Verification with Python and *py4hw*

David Castells-Rufas, Gemma Rotger
Universitat Autònoma de Barcelona (UAB)
Cerdanyola del Vallés, Spain
0000-0002-7181-9705, 0000-0002-9538-5278

David Novo
Laboratoire d'informatique, de robotique et de microélectronique de Montpellier (LIRMM)
Montpellier, France
0000-0002-5510-4152

***Abstract*—Classic hardware design languages, such as VHDL and Verilog, were born more than 30 years ago to improve the productivity of circuit design offered by manual schematic drawing. Over the years, the effort of Digital Hardware Design has shifted from circuit design to circuit verification. Although classic HDLs have been extended to adapt to this changing scenario, they have a hard time to cope with the complexity demanded at the design and verification stages. Radical changes incorporating high-level software programming languages in the design and verification processes seem inevitable. Accordingly, many new HDL proposals are already based on Python. In this paper, we present a novel Python-based HDL framework that tries to address some of the limitations of current Python-based HDLs with a special focus on verification and education.**

***Index Terms*—FPGA, Verilog, Python**

I. Introduction

The Python programming language has become very popular in the scientific community due to its simplicity, cross-platform support, easy learning curve, and a multitude of existing and ever-increasing functionality that is easily distributed through Internet-based setup infrastructures like PyPi and Conda. Hardware design languages (HDLs) like VHDL and Verilog were born as domain-specific languages to address the productivity limitations of manual schematic drawing. Modern digital circuits are generally designed using a compositional methodology, where circuits are subdivided into simpler interconnected blocks. This division is repeated until simple enough circuits, which are part of the platform primitives, are obtained.

The benefit of using a domain-specific language was justified when Object-Oriented (OO) languages were not mainstream, and it was not easy to express the compositional structure of hardware with them. Now, General-Purpose Programming Languages (GPPLs) can easily describe complex circuit descriptions by instantiating smaller blocks. The evolution of HDLs has moved the attention to the circuit verification features of the languages. We argue that, instead of trying to transform the domain-specific HDLs into GPPLs, it makes more sense to complement existing GPPLs to be able to describe Hardware blocks and provide a simulation infrastructure that allows complex verification. Several proposals have followed this approach. To name a few: SystemC [1] based on C/C++, JHDL [2] based on Java, and Chisel [3] based on Scala. We believe that some roadblocks hinder their wide adoption. While Java is still popular in the business domain, it is not so popular among the scientific and electrical engineering communities. Also, even with the relative success of Chisel, Scala is not a very popular language. On the other hand, while C/C++ is very popular in the scientific and electrical engineering communities, the language suffers from some limitations (e.g. poor introspection) that makes it less attractive to become the center of an HDL framework.

Despite the well-known performance limitations, Python still offers some compelling features to be the base of an HDL framework. There have been several proposals based on Python in the literature [4]–[9], but, they lack some features like visualization, interactive simulation, and, in some cases, they have an unclear separation between hardware design styles. These features are part of our proposed framework *py4hw* and could be incorporated into some of the other Python-based tools. As part of our commitment to the Open Hardware movement [10] and to improve education in Digital Hardware Design, we have made our tools available on both GitHub and PyPI under the GPL 3.0 license. We believe in the importance of open access to tools and resources for the advancement of the field, and we hope that our contributions will support the broader community of hardware designers and educators.

The organization of the paper is as follows: In Section II, we describe the main aspects of the *py4hw* framework. Special focus is given to the flexibility aspects of the framework in Section III and the visualization features in Section IV. Section V details the verification features of the framework. We describe the strength of the framework for education in section VI. Before concluding, Section VII analyzes related work and compares it with the current proposal.

II. *py4hw*

The functionality base of Python is far superior to that available in Verilog/SystemVerilog. Hence, using Python in *py4hw* provides an excellent foundation for elaborating complex hardware, defining of simulation stimuli, and developing verification testbenches. *py4hw* is specially optimized for the design of synchronous digital circuits, and it is influenced by JHDL [2].

The evolution of hardware design has consolidated three main design styles:

- Structural: description of blocs and their connections
- Register transfer level (RTL): description of the response of circuits at certain events (signal activation or clock edge)
- Sequential: Description of processes using a Von-Neumann-like approach where actions occur by the execution of an algorithm

Many Python-based HDLs mix design styles (as Chisel also does) or use non-intuitive methods to create hardware circuits, introducing unnecessary confusion to the designer. *py4hw* aims to differentiate between design styles and avoid language constraints that limit the freedom to create any circuit.

py4hw follows and Object Oriented (OO) design style. Every circuit (equivalent to Verilog module) is modeled by a Python class, which can be instantiated by other circuits. In Verilog, modules have two different sets of constructor-arguments, a mandatory set for the input and output wires, and an optional set for module parameters. In *py4hw*, the only mandatory constructor-arguments are the parent circuit and the name of the instance. *py4hw* maintains a hierarchical object model with parent-child relations. Each circuit instance contains a link to its parent, and a dictionary of children indexed by instance name. The top-level entity of the hierarchy is the `HWSystem` object. Thus, circuit constructors can have an arbitrary number of additional parameters, and the interface of the circuit (inputs and outputs) is built during runtime avoiding any static interface definition. The clock is implicit. All circuits have an assigned clock driver which is normally inherited from the parent circuit. However, multiple clock-drivers can be used, and there is support for gated clocks by using a special clock-driver object.

A fundamental class is the `Wire` class. An instance of the `Wire` class represents a signal that connects one source with one-or-more sinks. Wires have a width, a name, and a parent circuit that must be specified during creation. A significant difference with other HDLs is that wires are indivisible. Extracting specific bits from a wire, concatenating wires, or obtaining a range of wires from a wire require to use specific circuits. All circuits inherit from the class `Logic`, which provides some basic methods to create the circuit interface. Structural, RTL, and sequential design styles are supported in *py4hw*. The structural design style is based on the instantiation of objects from previously defined classes in the constructor of the class. The following code illustrates how to design a 1-bit Full Adder circuit with a structural design style.

```
class FullAdder(Logic):
  def __init__(self, parent, name, x, y, ci, s, co):
    super().__init__(parent, name)
    # interface definition
    x = self.addIn('x', x)
    y = self.addIn('y', y)
    ci = self.addIn('ci', ci)
    s = self.addOut('s', s)
    co = self.addOut('co', co)
    # internal wires
    w1 = self.wire('w1',1)
    w2 = self.wire('w2',1)
    w3 = self.wire('w3',1)
    # instances
    Xor2(self, 'g1', x, y, w1)
    Xor2(self, 'g2', w1, ci, s)
    And2(self, 'g3', w1, ci, w2)
    And2(self, 'g4', x, y, w3)
    Or2(self, 'g5', w2, w3, co)
```

py4hw also supports RTL design style by allowing to use the behaviour of the circuits using simple Python code. Two possible circuit types are supported: combinational and sequential. In both cases, the constructor of the class is used to define the interface and save their wires into the properties of the object. Combinational behavioral circuits must implement the `propagate` method. The following code illustrates how the previous Full Adder circuit can be described by using a combinational behavioural implementation.

```
class FullAdder(Logic):
  def __init__(self, parent, name, x, y, ci, s, co):
    super().__init__(parent, name)
    # interface definition
    ...

  def propagate(self):
    v = self.x.get() + self.y.get() + self.ci.get()
    self.s.put(v % 2)
    self.co.put(1 if v > 1 else 0)
```

Sequential behavioral circuits are defined by implementing the `clock` method. The following code illustrates how to implement a simple Finite State Machine (FSM) with two states.

```
class FSM(Logic):
  def __init__(self, parent, name, r):
    super().__init__(parent, name)
    self.r = self.addOut('r', r)
    self.state = 0

  def clock(self):
    if (self.state == 0):
        self.r.prepare(1)
        self.state = 1
    elif (self.state == 1):
        self.r.prepare(0)
        self.state = 0
```

Sequential design style can be implemented by combining Python coroutines with the implementation of `clock` method. With this approach, the circuit enters a coroutine on object creation which is automatically blocked after a first `yield` statement is reached. The `clock` method must call the `next` method of the coroutine so that the sequential process advances until the next `yield` statement is found. A similar approach is used in cocotb [11] and, from the programmer's perspective, is very similar to SystemC's sequential clocked processes (SC_CTHREAD), which used `sc_wait` statement instead of `yield`.

py4hw embeds a cycle-based simulator. The simulator is responsible to propagate the wire values across the circuit and update the state. Using behavioral models instead of structural ones is a known technique to speedup simulation. In the behavioral FullAdder example, the simulator requires a single method invocation to compute the outputs of the circuit. On the other hand, using structural design for the same circuit requires the invocation of the behavioural models of all its hierarchical descendants, which takes more time.

Simulation time can be advanced programmatically. This can be used in combination with state analysis to perform advanced verification and visualization strategies. This strategy was used in [12] to provide a rich visualization of the the state of a RISC-V processor while executing an compiled application. Several Python libraries (pyelftools, capstone, tkiner, ...) were used to provide a rich user experience.

py4hw generates Verilog from the circuit descriptions. Since the whole circuit hierarchy is maintained in memory, the RTL generation phase only has to traverse the circuit hierarchy and decide which method to apply to generate the equivalent Verilog code for each element of the hierarchy. Structural circuits are directly translated into structural Verilog code. Behaviourally modelled circuits are transpiled into Verilog. Their defining method (either `propagate` or `clock`) is analyzed using introspection to obtain the Abstract Syntax Tree (AST) of the method's source code. Several transformations are applied until the AST can be emitted as a valid Verilog. Calls to external libraries (such as NumPy) can not be used inside behavioural models that are synthesized. On the other hand, there is no restriction to use them on constructors of Structural circuits and behavioral models as simulation stimuli. Using Python's introspection eliminates the need for external libraries that would otherwise be necessary in languages like C/C++ to parse source code.

III. Circuit flexiblity

The interface of Verilog circuits is fixed by design. This limits the ability to build complex circuits and requires GP-PLs to assist EDA tools when such flexibility is needed. A often occurs when building a memory-mapped multiplexed bus to communicate a bus-master processor with a flexible number of bus-slave devices. Although the generative Verilog statements could assist in generating the module logic, the inflexibility of the language with respect to the circuit interface prevents the creation of a generic module. A work-around solution could be to design for the worst-case scenario by estimating the maximum number of inputs and leaving wires unconnected. However, this approach can lead to long and error-prone code due to the large number of wires. While SystemVerilog provides some relief with `interface` and `modport` statements, the issue of static interfaces is not fully resolved.

On the contrary, this use-case is extremely simple to implement in *py4hw*. First, because the circuit interface can be created during runtime by using `addIn` and `addOut` methods. Second, because semantically related signals can be grouped in any kind of Python data structure (such as a list, a dictionary, or a specific `Interface` class provided by *py4hw*) and passed to the circuit constructor. The following code illustrates a possible implementation.

```
class MultiplexedBus(Logic):
  def __init__(self, parent, name,
               master, slaves):
    super().__init__(parent, name)

    self.addInterfaceSink('master', master)
    for i, slave in enumerate(slaves):
      self.addInterfaceSource('slave{}'.format(i),
                              slave.if)
      addr = slave.addr
      ...
```

But interface flexibility is not the only required flexibility. Modern HDLs should also be able to support the manipulation of the circuit. Some intermediate low-level representations (IRs) have been proposed to describe hardware concepts. After the success of the LLVM infrastructure [13] in the Software world, it is expected that Hardware design flows can benefit from an infrastructure that allows the manipulation of the design descriptions in a number of optimization phases. The most relevant IRs are FIRRTL [14] and LLHD [15]. IRs provide a framework where manipulation is possible because in the original context (e.g. Verilog, C/C++, etc.) it was not. However, the in-memory object hierarchy of *py4hw* designs allows the unrestricted manipulation of the system making the IR phase unnecessary. Not all Python-based HDL frameworks support this flexibility. For instance, the MyHDL method to create circuits based on generators is not very adequate to implement this manipulation.

In *py4hw*, the circuit hierarchy can be easily traversed and manipulated, adding or removing ports from a circuit or creating new hardware. As an example, the following Python code manipulates a circuit to insert a global_enable signal. The intended goal is that the enable signal of every register found in the circuit hierarchy is controlled by the global_enable signal. The code checks whether each register uses the enable signal. If it does not, it adds it and connects it conveniently. Otherwise, it inserts a new `and2` gate combining the former enable signal with the global one. Furthermore, the global_enable signal must be included in the interface of all pertinent circuits in the hierarchy.

```
class GlobalEnableInserter:
  i = 0
  def transform(self, obj, gen):
    anyclk = False
    children = set(obj.children.values())

    for child in children:
      if (child.isClockable()):
        if (isinstance(child, Reg)):
          enable = child.e
          anyclk = True

          if (enable is None):
            child.addIn('e', gen)
          else:
            name = 'global_ena_{}'.format(self.i)
            self.i += 1
            new_enable = obj.wire(name)
            And2(obj, name, enable, gen, new_enable)
            child.reconnectIn('e', new_enable)
      elif (len(child.children) > 0):
        anyclk |= self.transform(child, gen)
    if (anyclk):
      obj.addIn('global_enable', gen)
      return True
    else:
      return False
```

IV. Visualization

Circuit visualization is important to understand the structure of a circuit and can be very helpful during validation, as some structural errors can be more easily identified by inspecting diagrams instead of source code. Logic diagrams are extensively used in education, and it also common practice to use them during the conceptualization of Hardware systems. Since complex circuits are designed in a hierarchical way, visualization tools require to be able to navigate the hierarchy and visualize the connection between all the elements of the system.

FPGA EDA tool manufacturers have included VHDL and Verilog circuit visualization features. However, visualization has to be elaborated by a process that analyzes the whole circuit and the user has little control to automate the generation of such diagrams. Other popular HDLs like Chisel [3], SystemC [1], or MyHDL [4] do not provide any circuit visualization feature.

On the other hand, JHDL [2] had rich visualization features, including interactive simulation with value annotation in the circuit schematic, which was very useful to debug circuit behavior. The HWT Python-based HDL library also includes visualization features.

py4hw has some generic goals regarding the visualization of circuits:

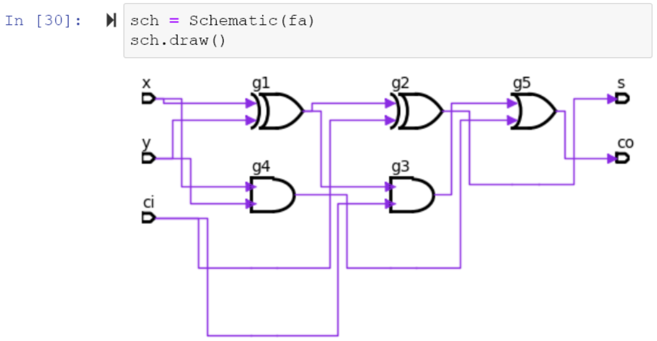
- Any part of the circuit hierarchy should be visualizable
- Circuit visualization should create a manipulable object hierarchy
- The user/designer should be able to control the details of the visualization process, from algorithmic aspects (like layout algorithms) to low-level details (such as colors and line widths).
- Visualization should allow multiple final targets, such as GUIs, images, or Jupyter Notebooks.

In *py4hw*, visualization is controlled by the `Schematic` class, which implements a circuit layout algorithm from scratch. An schematic diagram works with a circuit element from the hierarchy. It is not necessary to process all the circuit to generate a visualization from a node. The layout algorithm is responsible for placing input ports, child instances, and output ports of the circuit in the drawing canvas and route the connections among them.

The currently supported targets are `matplotlib` and `tkinter` canvas. The `Matplotlib` target is very useful to integrate circuit visualization if jupyter notebooks. Figure 1 depicts how a full adder circuit schematic is rendered in a jupyter notebook. The `tkinter` target is useful to visualize any circuit of the hierarchy in an interactive graphical user interface. This is actually used in the interactive simulation workbench that will be described in the following section.

V. Verification

Simulation is fundamental for verification. Simulation consist on evaluating how the system evolves over time under controlled conditions. Simulation sessions require a circuit design under test (DUT), some stimuli to inject to the circuit, and the ability to observe the behaviour of the DUT. A testbench consists of a combination of these three items.



```
In [30]:  sch = Schematic(fa)
          sch.draw()
```

Fig. 1. Schematic visualization of a combinational circuit in a Jupyter notebook

In Verilog, observability is based on printf-like messages and the generation of VCD trace files that record the activity of circuit wires for later inspection using waveform viewers such as the popular gtkwave tool. With SystemVerilog, additional formal methods like assertions were introduced.

In *py4hw*, stimuli can be created by several means. The most simple way is by using the `Constant` of `Sequence` circuits that allow to specify a constant value or a repetitive sequence of constant values respectively. For slightly more complex stimuli it is recommended to create a stimuli class using a sequential behavioral design style to specify inputs. When higher complexity stimuli is required, behavioural models using coroutines can be the best option.

The observability of wires in *py4hw* can be provided by multiple ways. First, wire information can be collected using the `Waveform` class and later displayed (see figure 2). The `Scope` class provides printf-like message outputs. Printf-like messages can also be embedded into user-defined circuits, either to debug the circuit creation phase in the constructor of structural circuits, or by providing info on the simulation progress on behavioral circuits. However, this approach is only recommended during early debuging phases and it is encouraged that these messages are removed from stable circuits to minimize the verbosity of systems during simulation. Assertions are easily supported either by using exceptions or assert statements.

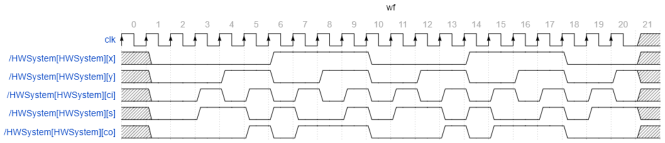
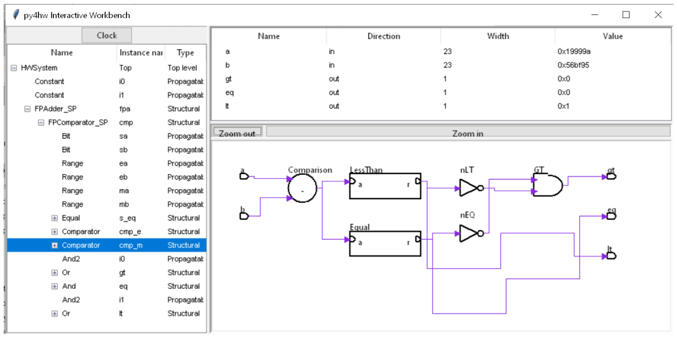


Fig. 2. Waveform visualization in Jupyter notebook

The simulator embedded in *py4hw* is cycle-based [16]. The simulator has a flattened view of all the primitive behavioral elements of the system. When the simulator is initialized



Name	Instance nar	Type
⊟ HWSystem	Top	Top level
Constant	i0	Propagatak
Constant	i1	Propagatak
⊟ FPAdder_SP	fpa	Structural
⊟ FPComparator_SP	cmp	Structural
Bit	sa	Propagatak
Bit	sb	Propagatak
Range	ea	Propagatak
Range	eb	Propagatak
Range	ma	Propagatak
Range	mb	Propagatak
⊞ Equal	s_eq	Structural
⊞ Comparator	cmp_e	Structural
⊞ Comparator	cmp_m	Structural
And2	i0	Propagatak
⊞ Or	gt	Structural
⊞ And	eq	Structural
And2	i1	Propagatak
⊞ Or	lt	Structural

Name	Direction	Width	Value
a	in	23	0x19999a
b	in	23	0x56bf95
gt	out	1	0x0
eq	out	1	0x0
lt	out	1	0x1

Fig. 3. Interfactive Workbench for a Floating Point adder circuit. We can simulate step by step. Left pane shows the hierarchy of the circuit. The top right pane has the interface of the circuit with the current signal values. The bottom right pane shows the schematic of the circuit.

all the elements are classified either being combinational or sequential. Combinational circuit chains are topologically sorted so that they can be correctly evaluated in sequence. This is possible as no asynchronous loops are allowed. In this way, no event propagation is needed, and circuits are only evaluated once. After evaluating combinational circuits, sequential circuits can be evaluated in any arbitrary order.

In *py4hw*, each wire can have an arbitrary width. In other HDL languages based on Java or C/C++, the wire width is limited by design because the underlying simulation infrastructure is using an integer type (64 bits, typically). This is not the case in Python as integers have arbitrary precision.

Modern design techniques require a systematic verification of all created circuits. In SystemVerilog, much effort was devoted to address complex verification. Moreover, the universal verification methodology (UVM), which is based on these SystemVerilog features, was proposed as an standard for the systematic verification of circuits. While unit testing, system testing, and continuous integration can be relatively new concepts to many Hardware designers, they have been very commonly used in complex software projects. Actually, there are several testing frameworks for Python. In *py4hw* we use `Pytest` to do unit testing. This allows to define systematic testing of all the circuits by using assertions, random stimuli, and equivalence checking.

After a bug is detected by using unit tests or system tests it is necessary to identify the source of the error. This can be done by analyzing waveforms in the classical post-mortem approach. However, an alternative method based on interactive simulation can be more efficient. *py4hw* provides an interactive workbench (see figure 3) in which the user/designer can navigate for the circuit hierarchy and view the circuit structure together with the values of the circuit interface. In addition to these methods custom state visualization can be built to transform the circuit state into a higher-level abstraction so that it is easy to identify.

VI. EDUCATION

Digital circuit design for FPGAs has historically required different tools from several vendors to design, synthesize, simulate and execute the designs. With the proliferation of Open-Source tools the cost aspect of the problem has been minimized, however, the user/designer still needs several tools to design and verify the circuit. Education at all levels is increasingly promoting self-paced courses and remotely accessible infrastructure. This is challenging in Digital Design courses due to the required infrastructure.

Jupyter notebooks are an effective method to combine documentation with software execution. Its use is being introduced in many engineering courses [17] either as part of self-paced courses or laboratory exercises. There a some consensus among digital design education community that students must visualize the circuits they can design with HDLs to fully understand the implications of their decisions. *py4hw* exercise materials have been prepared to use in a Digital Design Course so that the student can test his ability to create his/her circuits and automatically verify if they fulfill the requirements of the exercise. At the same time, a visualization of the created hardware is presented and the equivalent Verilog code is generated. Moreover, the student can run the exercises with the only requirement of a web browser, as the jupyter notebook using the *py4hw* framework can run on a `Binder` or `Collab` remote infrastructure.

VII. RELATED WORK

There are many proposals using Python for Hardware design. In Table I we list some of the most relevant.

MyHDL [4] is one of the first and well-known proposals to use Python for Hardware design. It was created in 2003 and has 858 starts in GitHub. Although it has an active community, the project seems to be loosing steem and entering a decay phase as can be observed by the last PyPi release date. One of the drawbacks of MyHDL is that component creation was based on generator functions. Each module is described as a function that creates logic entities that are returned as list of objects. Decorators are used to ease the definition of such functions. Hierarchy was hard to obtain in the initial releases. New releases are moving to an approach were hierarchy is maintained, but the momentum of the framework will make the change slow. PyRTL has a similar issue with its structure, as it opts not to use classes for circuit creation and instead aims to simplify usage by concealing hardware creation details.

According to Github starts, the most popular Python-based frameworks are Migen and its derived Amaranth. Their drawback are that they promote a syntax that simultaneously combines different design styles in the source code, which is (from our experience) a source of confusion for many undergraduate students. Moreover, according to the last PyPi release date, development halted in 2019 for Migen, and 2021 for Amaranth.

The project with the most recent PyPi releases is PyMTL3 [8]. It provides different clear design styles which are annotated with function decorators. It also maintains a hierarchy that allows applying transformations as proposed by IR tools. In addition, it is actively used in hardware design courses at Cornell University and Boston University. An interesting approach is followed by PyLog [9], which provides automatic deployment in some supported FPGAs. However, the language hides the details of hardware structure making it less useful for education. Moreover, it is not distributed through PyPi. HWT [7] covers many aspects of Hardware design. Its main drawback might be the fragmentation of the functionality in several repositories.

None of the analyzed frameworks provide circuit visualization, except for HWT. HWT has a good visualization tool that can be inserted in Jupyter. It is based on the ELK (Eclipse Layout Kit), which is based on Java and is automatically transpiled in Javascript using GWT. Consequently, none of them provide a GUI for interactive simulation, which is very useful for education.

TABLE I
OPEN SOURCE PYTHON-BASED HDL FRAMEWORKS

Framework	Launched	Last PyPi Release	GitHub Repository	Stars
MyHDL [4]	2003	6/2019	myhdl/myhdl	886
PyRTL [6]	2015	9/2021	UCSBarchlab/PyRTL	173
HWT	2016	7/2021	Nic30/hwt	160
Migen	2019	11/2019	m-labs/migen	987
PyMTL3 [8]	2020	5/2022	pymtl/pymtl3	254
PyLog [9]	2021		hst10/pylog	39
Amaranth	2021	12/2021	amaranth-lang/amaranth	1.1k

In addition to these frameworks there are other related Python-based projects with interesting results. PyVerilog [18], available at PyHDI/Pyverilog is not an HDL framework but it is a library for Verilog analysis providing the ability to parse and manipulate Verilog code.

A very relevant work is cocotb [11], with 1.1k stars in GitHub and a last PyPi release in February 2022. Cocotb focuses on testbench generation from Python to inject stimuli into standard HDL projects. To do so, it basically provides a simulation engine that interacts with external simulators (such as QuestaSim, Verilator, etc) and uses Python coroutines to model stimuli generators.

VIII. CONCLUSION

Classic HDL languages are incorporating features of standard programming languages to cope with the requirements of hardware design and verification. Alternatively, hardware creation with software programming languages might be simpler and opens the door to many new engineers to create Hardware.

Currently, Python is one of the most popular programming languages with a rich ecosystem of tools and libraries and a big developer community. There have been several proposals to use Python for hardware design.

Many of them take a very high-level description approach hiding the details of hardware design. In digital design education courses, it is necessary to be able to control the created hardware with good detail and be always clear about what design style we are using. It is also fundamental to visualize the results as soon and as often as possible. Even if using high-level synthesis, the transformations must be identifiable and explainable to students.

py4hw tries to address these issues by providing a low-level Python library for the compositional creation of digital circuits, their verification and visualization using modern tools such as interactive GUIs and jupyter notebooks. By eliminating the need for third-party software installations and offering early visualization of designed circuits, students have less entry barriers and iterate faster with digital hardware design.

REFERENCES

[1] P. R. Panda, “Systemc: a modeling platform supporting multiple design abstractions,” in *Proceedings of the 14th international symposium on Systems synthesis*, 2001, pp. 75–80.

[2] P. Bellows and B. Hutchings, “Jhdl-an hdl for reconfigurable systems,” in *Proceedings. IEEE symposium on FPGAs for custom computing machines (Cat. No. 98TB100251)*. IEEE, 1998, pp. 175–184.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: constructing hardware in a scala embedded language,” in *DAC Design automation conference 2012*. IEEE, 2012, pp. 1212–1221.

[4] J. Decaluwe, “Myhdl: a python-based hardware description language.” *Linux journal*, no. 127, pp. 84–87, 2004.

[5] “Migen (milkymist generator), a python toolbox for building complex digital hardware.” [Online]. Available: https://github.com/m-labs/migen

[6] D. Mirza, D. Dangwal, and T. Sherwood, “Pyrtl in early undergraduate research,” in *Proceedings of the Workshop on Computer Architecture Education*, 2019, pp. 1–8.

[7] “Hwtoolkit (hwt) the library for hardware development in python.” [Online]. Available: https://github.com/Nic30/hwt

[8] S. Jiang, P. Pan, Y. Ou, and C. Batten, “Pymtl3: a python framework for open-source hardware modeling, generation, simulation, and verification,” *IEEE Micro*, vol. 40, no. 4, pp. 58–66, 2020.

[9] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. Hwu, “Pylog: An algorithm-centric python-based fpga programming and synthesis flow,” *IEEE T COMPUT*, vol. 70, no. 12, pp. 2015–2028, 2021.

[10] F. Hannig and J. Teich, “Open source hardware,” *Computer*, vol. 54, no. 10, pp. 111–115, 2021.

[11] B. J. Rosser, “Cocotb: a python-based digital logic verification framework,” in *Micro-electronics Section seminar*. CERN, Geneva, Switzerland, 2018.

[12] A. Hammani Abbasi, “Implementació d’un entorn interactiu educatiu per a la docència d’arquitectures risc-v,” 2021.

[13] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[14] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson *et al.*, “Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 209–216.

[15] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “Llhd: A multi-level intermediate representation for hardware description languages,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 258–271.

[16] S. Palnitkar and D. Parham, “Cycle simulation techniques,” in *Proceedings. 1995 IEEE International Verilog HDL Conference*. IEEE, 1995, pp. 2–8.

[17] A. Cardoso, J. Leitão, and C. Teixeira, “Using the jupyter notebook as a tool to support the teaching and learning processes in engineering courses,” in *International Conference on Interactive Collaborative Learning*. Springer, 2018, pp. 227–236.

[18] S. Takamaeda-Yamazaki, “Pyverilog: A python-based hardware design processing toolkit for verilog hdl,” in *International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 451–460.

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 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.

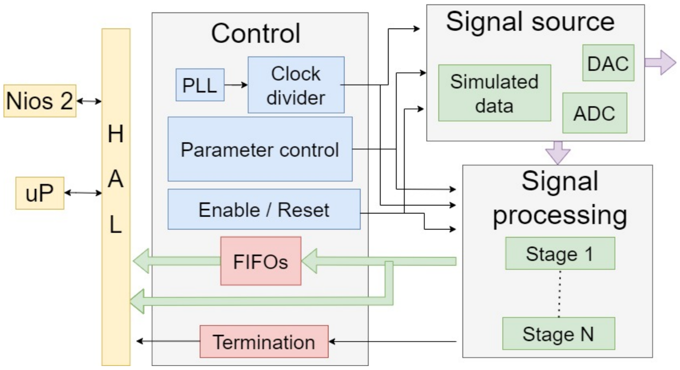


Fig. 1. Design's general structure.

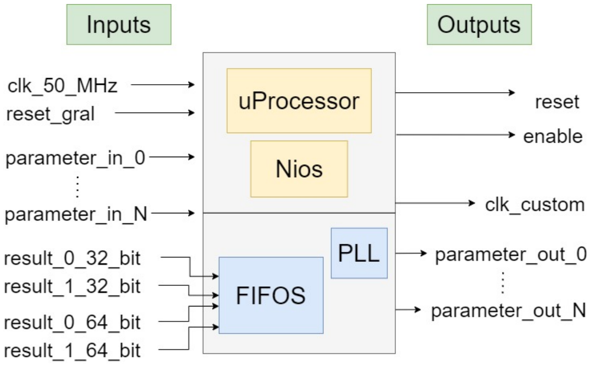


Fig. 2. Control module.

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],

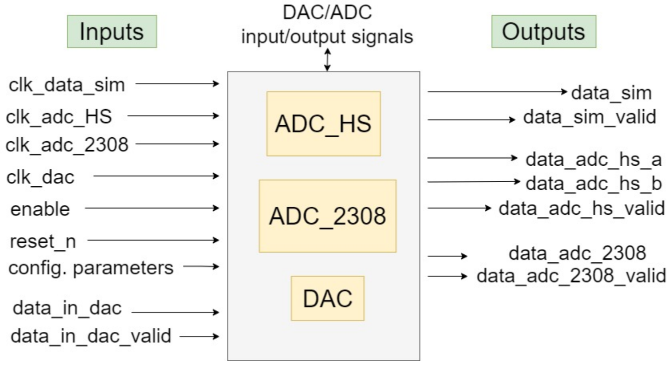


Fig. 3. Signal source module.

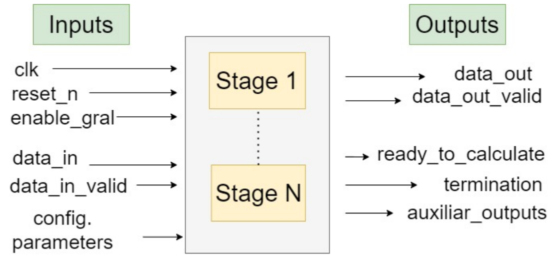


Fig. 4. Signal processing module.

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 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:

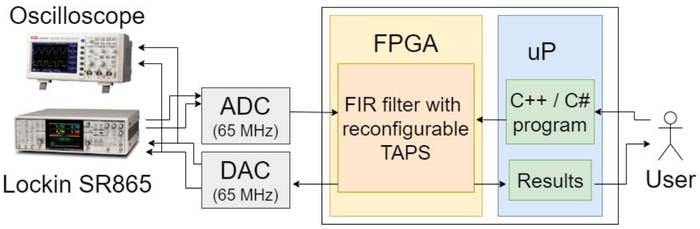


Fig. 5. Testing setup.

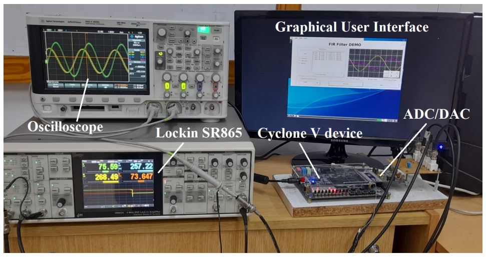


Fig. 7. Testing Setup.

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.

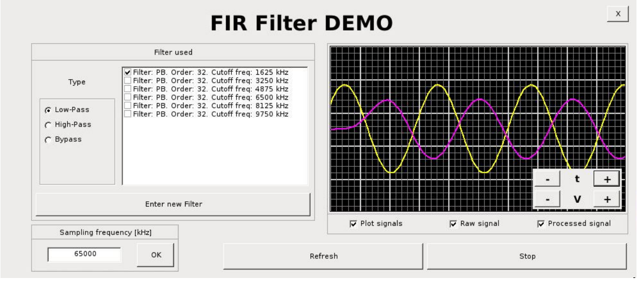
IV. Results

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.


FIR Filter DEMO

Filter used

Type

● Low-Pass
○ High-Pass
○ Bypass

☑ Filter: PB. Order: 32. Cutoff freq: 1625 kHz
□ Filter: PB. Order: 32. Cutoff freq: 3250 kHz
□ Filter: PB. Order: 32. Cutoff freq: 4875 kHz
□ Filter: PB. Order: 32. Cutoff freq: 6500 kHz
□ Filter: PB. Order: 32. Cutoff freq: 8125 kHz
□ Filter: PB. Order: 32. Cutoff freq: 9750 kHz

Sampling frequency [kHz]: 65000

☑ Plot signals ☑ Raw signal ☑ Processed signal

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 $| H(f) |$ of both filters was measured with the SR865, and then compared with the theoretic transference of the filters. The results are shown in Fig. 10, and Fig. 11.

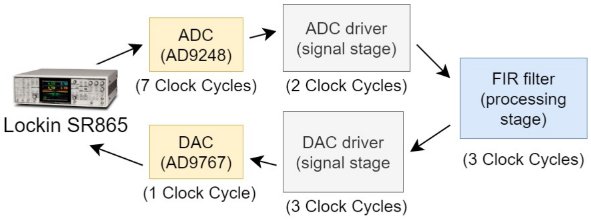


Fig. 8. Signal path.

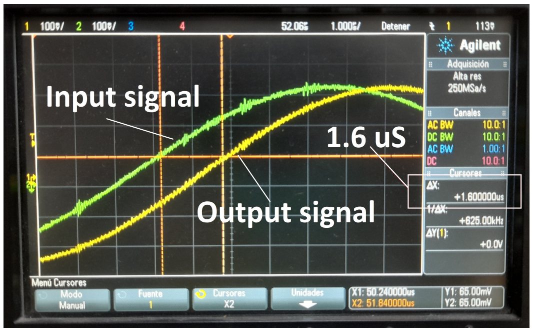


Fig. 9. Timing delay measurment.

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)

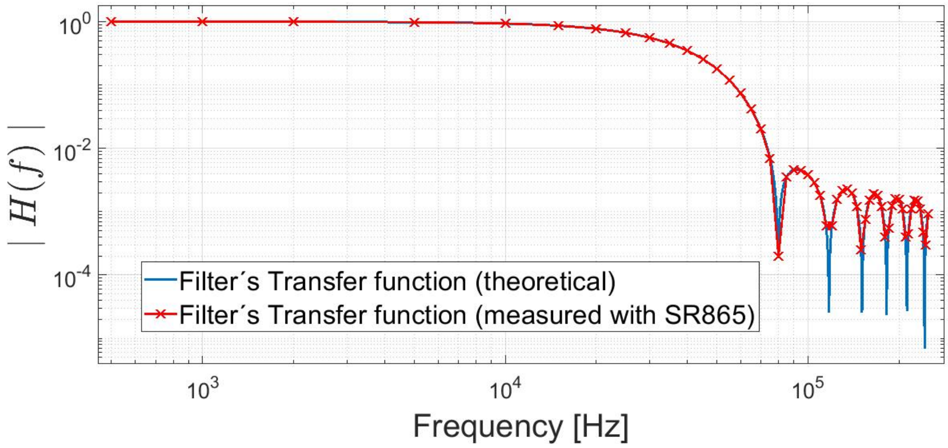


Fig. 10. Low-pass filter's transference.

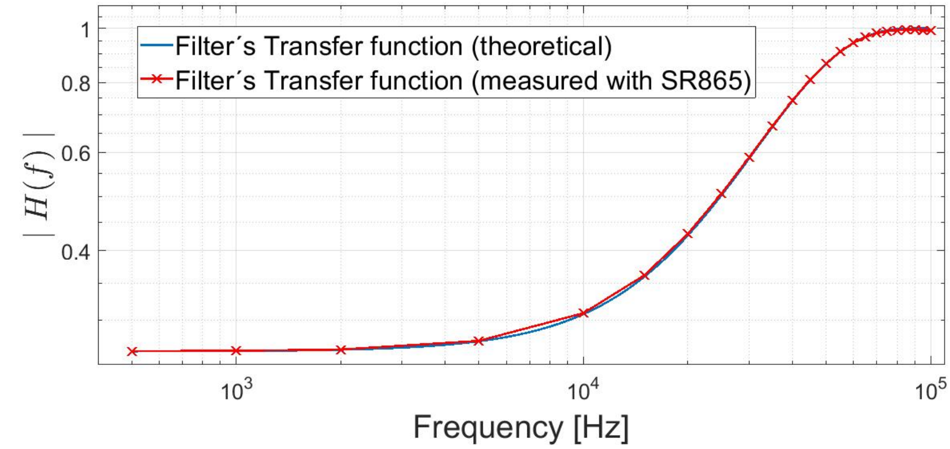


Fig. 11. High-pass filter's transference.

[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).

Turbo-Código seguro mediante *Interleaver* aleatorio variable en el tiempo

Raúl Eduardo Lopresti, Maximiliano Antonelli, Jorge Castiñeira Moreira y Luciana De Micco
Instituto de Investigaciones Científicas y Tecnológicas en Electrónica (ICYTE)
Facultad de Ingeniería, Universidad Nacional de Mar del Plata (FI-UNMdP)
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

***Abstract*—Los nuevos estándares de comunicaciones ponen el foco tanto en una baja probabilidad de error como en la seguridad de la transmisión. Estas propiedades son clásicamente excluyentes y requieren de varias etapas de procesado de la información a transmitir. En el caso de los códigos Turbo es posible aumentar la seguridad de la transmisión mediante la variación temporal del bloque Interleaver, manteniendo la probabilidad de error. En este trabajo, se propone un sistema de cripto-codificación y se lo estudia en cuanto a probabilidad de error, encriptamiento y complejidad del sistema final. Se presenta también un diseño preliminar del sistema en un dispositivo digital, para verificar su factibilidad.**

***Index Terms*—Códigos Turbo, Interleaver, Encriptamiento, Seguridad.**

I. Introducción

Garantizar la confiabilidad e integridad de los datos transmitidos en un canal ruidoso es un tema de mucho interés en las comunicaciones digitales. Por otra parte, la seguridad de la transmisión también es un factor clave. Tradicionalmente los procesos de codificación y encriptamiento se realizan en operaciones separadas y en general el proceso de encriptamiento degrada la confiabilidad de los datos incrementando la probabilidad de error del sistema [1].

En el caso de la codificación para el control de error los Códigos Turbo (CT) son ampliamente utilizados debido a su simplicidad y eficiencia. Además, presentan la ventaja de ser flexibles en cuanto a su arquitectura, permitiendo incluir alternativas que agreguen un nivel de seguridad.

Recientemente han surgido trabajos que proponen la incorporación de seguridad en la codificación, ya que los nuevos estándares de comunicación tienden a priorizar estas características. En [2] se propone un esquema de cifrado simétrico para mejorar la seguridad de la transmisión. Los autores plantean un sistema que realiza en el codificador un mezclado (*interleaving*) de los bits de información e intercalado (*puncturing*) de los bits de paridad. La forma de realizar los procesos de *interleaving* y *puncturing* está controlada por una clave privada de encriptación simétrica.

En [3] los autores proponen un método de cifrado y codificación conjunto llamado CFB-AES-TURBO, el cual combina el cifrado AES (Advanced Encryption Standard) y la codificación Turbo. En [4] se propone un codificador Turbo modificado en el que los bloques constitutivos del codificador son máquinas secuenciales de estado finito (FSSM, Finite State Sequential Machine) en las que se inserta una función no lineal la cual varía sus coeficientes en el tiempo. Mediante esta configuración, los autores consiguen incrementar la seguridad del sistema Turbo manteniendo su rendimiento (*performance*).

En este trabajo presentamos un codificador Turbo que mejora su seguridad mediante la variación en el tiempo de su Interleaver, éste es generado en forma aleatoria mediante una regla. La elección de la regla a utilizar se realiza teniendo en cuenta que debe generar secuencias con buenas propiedades estadísticas y a la vez ser simple, para facilitar su implementación. Se estudia la utilización de diferentes generadores de números aleatorios (PRNG, Pseudo Random Number Generator) con períodos de diferentes longitudes. Se busca que la Probabilidad binaria de error (P_{be}) no se degrade, que la aleatoriedad de la salida aumente y que la complejidad del sistema a implementar no se incremente en forma significativa.

II. Interleaver variable en el tiempo

En la Fig. 1 se muestra la estructura de un codificador Turbo, el mismo cuenta con dos FSSM, donde una recibe los datos en forma directa y la otra a través de un Interleaver. Luego, un bloque se encarga de elegir entre la salida de cada una de estas máquinas (puncturing). Además, como se trata de una codificación sistemática, el dato de información se transmite también en forma directa. Finalmente, los datos generados son formateados a un esquema binario polar. En este trabajo se propone utilizar un bloque Interleaver que varíe en el tiempo. Éste se compone de un PRNG el cual entregará N números en cada transmisión a partir de una semilla inicial, siendo N el tamaño del Interleaver. Se sugiere implementar el PRNG con un generador simple como un LFSR (Linear Feedback Shift Register) o un mapa caótico, con el objetivo de que su implementación en hardware requiera una mínima cantidad de recursos adicionales. El receptor debe tener la información de la semilla empleada para generar el primer Interleaver, ésta y el conocimiento del mapa o PRNG utilizado son la llave del sistema.

Existen diferentes formas de realizar la intercalación de los datos [5], [6]. Una de las más sencillas es usando una matriz bidimensional con R filas y C columnas tales que $RC \geq N$, $R \geq 2$ y $C \geq 2$, en la que se escriben los

Este trabajo fue parcialmente financiado por el CONICET (Consejo Nacional de Ciencia y Tecnología), la Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT, PICT2019-03024).

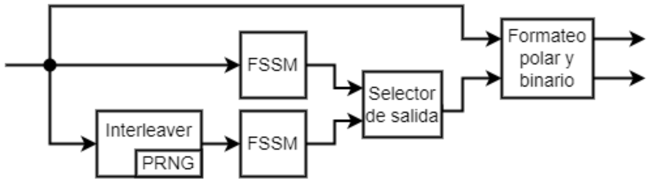


Fig. 1. Esquema simplificado de un codificador Turbo, el bloque de Interleaver se genera a partir de un PRNG y varía en cada transmisión.

datos fila por fila y luego se leen columna a columna. Una variación de este método consiste en realizar intercambios entre filas y entre columnas antes de leer los datos para obtener una secuencia más mezclada. En este trabajo, los datos se escriben secuencialmente en la posiciones que resultarían de realizar dichas permutaciones c, reduciendo así la latencia en el proceso de codificación. Además, para aumentar la seguridad de la transmisión se realizan intercambios diferentes y aleatorios en cada bloque de N datos. Estos métodos se ejemplifican en la Fig. 2.

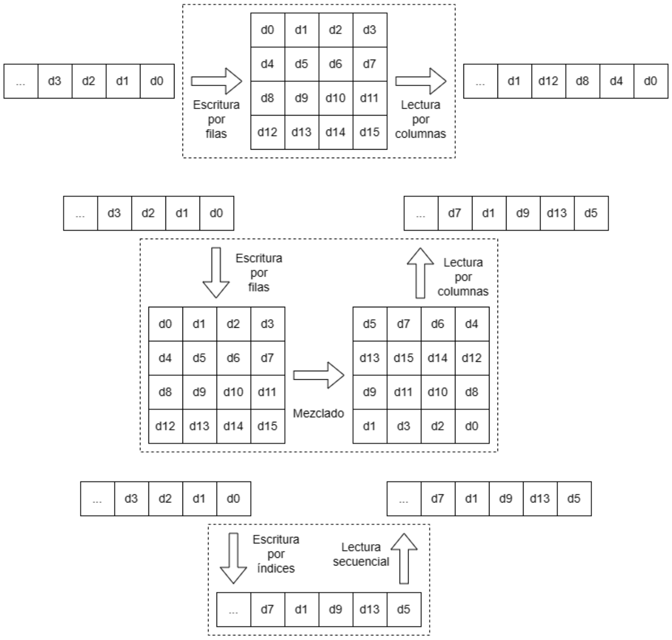


Fig. 2. Esquemas de algunos métodos de un intercalador. En la parte superior se ejemplifica una matriz clásica; en el medio, una matriz con intercambios de filas y columnas; y en la parte inferior, una variante de la anterior en la que los datos se escriben directamente en las posiciones que resultan del mezclado.

III. Simulaciones

Para obtener resultados preliminares se realizó un programa en $Matlab$ que simula la transmisión, el canal y el receptor. En el esquema propuesto, se genera un Interleaver aleatorio para cada transmisión. La llave para la codificación y la decodificación se comparte en el principio de la transmisión y luego se va generando un nuevo Interleaver para cada paquete transmitido. Por ejemplo, en nuestro caso, se generaron paquetes con datos aleatorios de tamaño $N = 400$. Estos datos ingresan a las FSSM las cuales operan en un campo de Galois GF(4). Una de las FSSM recibe los datos en forma directa y la otra máquina, a través de un Interleaver generado localmente. Luego, se simula un canal AWGN (Additive White Gaussian Noise) y se decodifican los datos con un Interleaver y de-Interleaver generado localmente en el receptor mediante el PRNG a partir de la misma semilla. Se repite este proceso para una cantidad de $g = 1000$ transmisiones y se promedian los resultados. Finalmente, se compara la probabilidad binaria de error P_{be} con la obtenida para el mismo esquema pero con la solución clásica del Interleaver en bloque, en donde los datos se almacenan por filas en una matriz de $M \times M$ (20×20 en nuestro caso) y se leen por columnas [7].

En la Fig. 3 mostramos estos resultados, en este caso el codificador se implementó en GF(2) para mayor simplicidad y velocidad en las simulaciones. Estos resultados son fácilmente extrapolables a otros GF(n). Se simularon tres tipos distintos de Interleavers:

- *Aleatorio*. El Interleaver es generado en cada transmisión con un generador aleatorio. Este vector se ordena de menor a mayor y se registran los intercambios en los índices entre el vector sin ordenar y el vector ordenado. Este vector de índices es el Interleaver.
- *Logístico*. El Interleaver es generado en cada transmisión con un mapa caótico logístico.
$$x(j+1) = 4x(j)(x(j)-1) \qquad (1)$$
con condición inicial $x(0) = 0,1$. Luego se repite el procedimiento de ordenar y registrar los índices como en el caso del aleatorio.
- *LFSR*. Se genera un vector binario aleatorio de longitud $L = 16N$ bits con el polinomio generador $x^{16} + x^{15} + x^{13} + x^4 + 1$. En el vector resultante se agrupan palabras de 16 bits y se convierten a decimales. Luego se repite el procedimiento de ordenar y registrar los índices como en el caso del aleatorio.
- *Bloques*. Se muestran también los resultados del esquema clásico de Interleaver en bloques como referencia.

Podemos ver que el esquema de codificación con Interleaver aleatorio resulta en menos errores de transmisión, por lo que se mejora la P_{be}. El caso que más se acerca al esquema clásico es el mapa logístico, esto se debe a que este mapa presenta patrones que resultan en un Interleaver mal mezclado.

IV. Implementación en Hardware

En la Fig. 4 se presenta un diagrama del circuito diseñado e implementado en FPGA para estudiar las factibilidades y características del esquema propuesto. Consta de los siguientes tipos de bloques:

- PRNG → Provee un número de manera pseudoaleatoria en cada ciclo de reloj.
- SIPO (Serial-Input Parallel-Output) → Registro de desplazamiento con entrada serie y salida paralela. Sirve de interfaz en las etapas de pipeline.
- EIPO → Es un SIPO en el que las escrituras se producen en posiciones arbitrarias en lugar de incrementales.

- PISO (Parallel-Input Serial-Output) → Registro de desplazamiento con entrada paralela y salida serie. Sirve de interfaz en las etapas de pipeline.
- PIRO. Es un PISO en el que los datos rotan en el registro en lugar de ser descartados.
- Sorting → Ordena los números y provee índices de las nuevas posiciones.
- FSSM → Realiza la codificación. En particular, el bloque e_0 FSSM proporciona además su estado de trellis por medio de la señal s.
- Ending → Realiza la terminación del trellis pertinente a los TCs.
- Indexing → provee el índice para el bloque EIPO.
- Selector → Realiza el puncturing del CT. En cada ciclo de reloj escribe alternativamente a su salida una de sus entradas.
- Control (no se presenta en el diagrama por legibilidad del mismo) → Sincroniza el sistema mediantes señales de habilitación y control.

El diseño cuenta con 4 etapas pipeline identificadas en la diagrama con S_0, S_1, S_2 y S_3. En las que se realizan las tareas que se describen a continuación:

1) S_0 → Se generan y registran C y R números aleatorios de forma concurrente. La cantidad de ciclos requeridos es igual $T_0 = \max(C, R)$.
2) S_1 → Se reordenan estos dos grupos de números de manera ascendente y se registran sus respectivas posiciones, las que serán los nuevos índices de filas i_R y columnas i_C para la intercalación de los datos. La cantidad de ciclos requeridos T_1 depende del algoritmo de ordenamiento que se utilice.
3) S_2 → Se realizan tres tareas en simultáneo durante $T_2 = N$ ciclos de reloj:
    - Se registra u, serie de entrada r más datos de terminación de trellis.
    - Se produce y almacena la secuencia código e_0 a partir de u.
    - Se crea u_i, la versión intercalada de u, guardando los datos en las posiciones dictadas por el índice i_u en base a los índices i_R e i_C.
4) S_3 → Se hacen dos tareas en paralelo durante $T_3 = N$ ciclos de reloj:
    - Se produce la secuencia código e_1 con u_i y la serie e que resulta de alternar entre e_0 y e_1 en cada ciclo de reloj. Es decir $e = (e_0(0), e_1(1), e_0(2), e_1(3), ...)$.
    - Se propaga u.

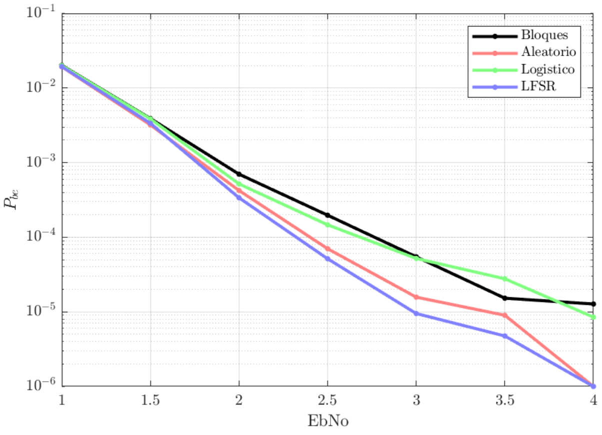


Fig. 3. Resultados de simulaciones. Puede verse que el Interleaver implementado con un LFSR es el de mejor *performance* para canal Gaussiano.

Particularmente, en este trabajo se experimentó con la siguiente configuración:

- $N = 400$, por tanto, $T_2 = T_3 = N$
- $R = C = \sqrt{N} = 20$, por ende, $T_0 = 20$
- se usaron LFSR de 32 bits para ambos PRNG.
- para el ordenamiento se empleó el algoritmo de ordenamiento paralelo bitonic modificado [8] para iterar sobre un solo nivel reduciendo la cantidad de hardware necesaria para su implementación sin aumentar la cantidad de ciclos requeridos de procesamiento. En este caso $T_1 = 15$.

V. Conclusiones y Trabajo Futuro

Los resultados preliminares mostraron que es posible realizar una cripto-codificación mediante un Interleaver aleatorio variante en el tiempo. Las simulaciones realizadas hasta el momento han presentado buenos resultados en cuanto a la probabilidad binaria de error obtenida. La implementación en hardware propuesta no requiere de una gran cantidad de recursos extra, lo que indica que es factible su implementación. Se prevé verificar el funcionamiento del sistema propuesto empleado distintos mapas caóticos, y LFSR de diferente período. Se testearán las salidas del codificador Turbo a entrada nula mediante los test de aleatoriedad NIST [9] y distintos cuantificadores de aleatoriedad. Se implementará en hardware tanto el codificador como el decodificador para evaluar la factibilidad del sistema y la *performance* presentada.

References

[1] R. E. Lopresti and J. C. Moreira, “Hardware-level secure coding,” *IEEE Embedded Systems Letters*, 2023.

[2] G. Zhu, D. Chen, C. Zhang, and Y. Qi, “Secure turbo-polar codes information transmission on wireless channel,” in *2021 IEEE 15th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. IEEE, 2021, pp. 116–121.

[3] S. Jeon and J. P. Choi, “Cfb-aes-turbo: joint encryption and channel coding for secure satellite data transmission,” in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–7.

[4] L. De Micco, D. Petruzzi, H. A. Larrondo, and J. Castiñeira Moreira, “Randomness of finite-state sequence machine over gf (4) and quality of hopping turbo codes,” *IET Communications*, vol. 7, no. 9, pp. 783–790, 2013.

[5] A. A. Jassim and W. A. Hadi, “Performance comparison of proposed interleaver with different types for parallel turbo code,” *Journal of Engineering and Sustainable Development*, vol. 2018, pp. 143–156, 7 2018.

[6] D. Prabhavati and K. Shantanu, “Ber analysis of turbo code interleaver,” *International Journal of Computer Applications*, vol. 126, pp. 1–4, 9 2015. [Online]. Available: https://www.ijcaonline.org/research/volume126/number14/bahirgonde-2015-ijca-906278.pdf

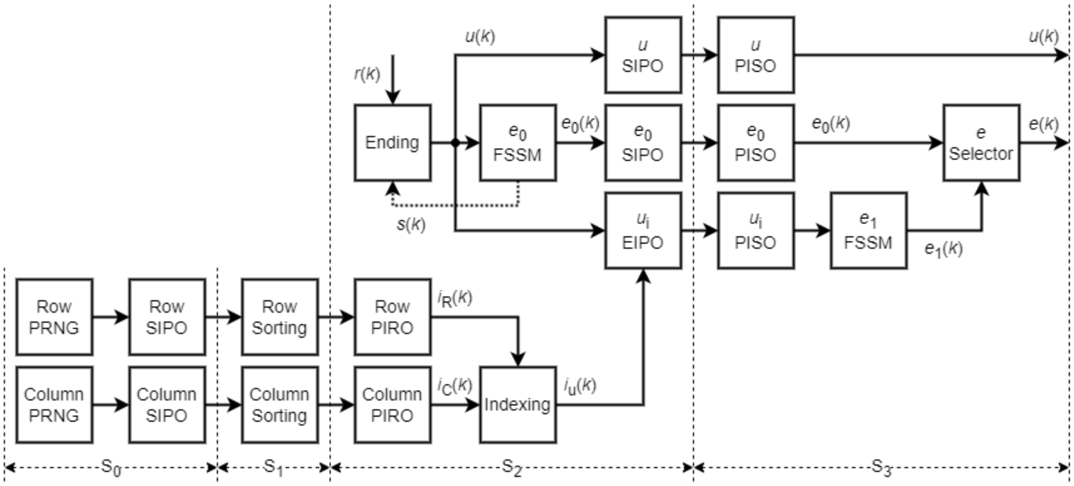


Fig. 4. Diagrama del diseño implementado en una FPGA para evaluar el esquema propuesto.

[7] R. E. Lopresti, M. Antonelli, J. Castiñeira Moreira, and L. De Micco, "Codificación segura a nivel de hardware," in *2022 Congreso Argentino de Sistemas Embebidos CASE*. ACSE - Asociación Civil para la investigación, Promoción y Desarrollo de Sistemas Eléctricos Embebidos, 2022, p. 27.

[8] K. J. Liszka and K. E. Batcher, "A generalized bitonic sorting network," in *1993 International Conference on Parallel Processing-ICPP'93*, vol. 1. IEEE, 1993, pp. 105–108.

[9] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," Booz-allen and hamilton inc mclean va, Tech. Rep., 2001.

Evaluation of dense and sparse linear algebra kernels in FPGAs

Federico Favaro*, Ernesto Dufrechou†, Juan P Oliver* and Pablo Ezzatti†
*Instituto de Ingeniería Eléctrica (IIE)
Universidad de la República (UDELAR), Montevideo, Uruguay
Email: {ffavaro, jpo}@fing.edu.uy
†Instituto de Computación (INCO)
Universidad de la República (UDELAR), Montevideo, Uruguay
Email: {edufrechou, pezzatti}@fing.edu.uy

***Abstract*—Numerical Linear Algebra (NLA) is a research field that has been characterized by the use of kernel libraries that are de facto standards. One of the most notable examples, particularly in the HPC field, is the Basic Linear Algebra Subroutines (BLAS). Most BLAS operations are fundamental to many scientific algorithms because they typically constitute the most computationally expensive stage. For this reason, numerous efforts have been made to optimize these operations on various hardware platforms. There is growing concern in the HPC world about power consumption, making energy efficiency an increasingly important quality when evaluating hardware platforms. Due to their greater energy efficiency, Field-Programmable Gate Arrays (FPGAs) are now emerging as an interesting alternative to other hardware platforms for accelerating this type of operations. Our study focuses on the evaluation of FPGAs for dense and sparse NLA operations. Specifically, we explore and evaluate the available options for two of the most representative BLAS kernels, i.e. gemv and gemm and one of the most important sparse linear algebra kernel, ie. SpMV. The experimental evaluation is conducted on the Alveo U50 and U280 accelerator cards from Xilinx and an Intel Xeon Silver multicore CPU. Our findings show that even in kernels where the CPU performs better in terms of runtime, the FPGA counterpart is more energy efficient.**

***Index Terms*—dense and sparse numerical linear algebra, energy-efficiency, FPGA**

I. Introduction

Numerical Linear Algebra (NLA) traverses several disciplines of science and engineering, such as artificial intelligence, optimization, computer graphics and control, and is a field of great importance in scientific computing. Solving problems in these areas often includes, as the most computationally expensive step, basic NLA operations such as the general matrix-matrix multiplication (GEMM), general matrix-vector multiplication (GEMV) or the operation of multiplying a sparse matrix by a vector (SpMV) [1], [2].

For the SpMV, perhaps the best known context is the resolution of linear systems of equations by iterative methods. In these cases, the method involves the repeated use of the SpMV operation, maintaining the same (sparse) matrix but varying the vector to be multiplied. The importance of this sparse NLA kernel in the community has motivated a large amount of work focused on its optimization and performance improvement.

The importance of NLA is also supported by the strong presence of basic algebra operations in the most widespread benchmarks. One of the most notorious is the Linpack benchmark [3] that is employed to define the Top500 list [4]. This benchmark is based on the LU-factorization to compute the peak performance reached by a specific combination of a hardware platform and software implementations. The LU-factorization is part of the LAPACK specification [5] and typically these kinds of methods are built over BLAS kernels. This philosophy of developing several layers of kernels specifications has guided the dense NLA landscape since the 70s. Firstly, with the BLAS-1 specification [6], later with BLAS-2 [7] and BLAS-3 [8], and subsequently LAPACK and ScaLAPACK [9].

Nowadays, energy consumption and power dissipation have become two of the main limitations in hardware design. This has been causing a growing concern in the HPC community [1], due to the difficulty of developing new hardware to sustain the increasing demand for computing, the economic cost of electricity, and the environmental impact. In particular, in NLA the focus is in the energy consumption required to compute the different kernels [10]–[12].

The concern about energy has prompted, in the last decade, a search to develop new specialized hardware architectures.Field-Programmable Gate Arrays (FPGA) have had a great evolution in recent years, positioning themselves in the ecosystem of reconfigurable heterogeneous devices as one of the most energy efficient alternatives to solve certain types of parallel algorithms. In general, FPGA based accelerators offer less raw computing power and memory bandwidth than other heterogeneous platforms like GPUs. But the gap is narrowing as manufacturers are making big efforts improve FPGA capabilities. Moreover, given their lower power consumption, there is an active topic of research for energy efficiency on these platforms.

The classic FPGA design strategy involves the use of low-level hardware description languages (HDL) such as VHDL or Verilog. These impose different programming models than standard software languages, with longer development times and complex debugging. To overcome this disadvantage, manufacturers are pushing to adopt High-Level Synthesis (HLS) languages like C/C++ and OpenCL. This allows for a more significant adoption of FPGAs as hardware accelerators by the software community.

In a previous work [13] we studied the potential of modern datacenter oriented FPGAS to address NLA operations. In particular, we evaluated available open-source implementations for two of the most representative kernels of BLAS, i.e. GEMM and GEMV. The experimental evaluation was carried out in an Alveo U50 accelerator card from Xilinx and, for comparison purposes, on a Intel Xeon Silver multicore CPU. This work extends our previous efforts by adding a new platform, the Alveo U280 accelerator card, and also by including a fundamental kernel of sparse linear algebra, SPMV. All the evaluated kernels were developed in HLS C++ specifically for Xilinx devices.

The rest of the paper is structured as follows. In Section II we summarize the NLA operations involved in this study. Later, in Section III, we describe the studied kernels implementations. This is followed by the experimental evaluation in Section IV. Finally, in Section V we present the conclusions and lines of future work.

II. Numerical Linear Algebra

In this section we briefly introduce the BLAS specifications with two of the most representative operations, GEMM and GEMV. From the sparse linear algebra perspective, we present the SPMV operation.

A. BLAS

Numerical Linear Algebra (NLA) is characterized by the use of standardized kernel-libraries. A remarkable example is the Basic Linear Algebra Subroutines (BLAS) [14], a library that has become essential for HPC because of its portability, efficiency, and availability. BLAS is organized into three levels. Level 1 involves scalar, vector and vector-vector operations, level 2 includes matrix-vector operations, and level 3 performs matrix-matrix operations. BLAS has become one of the main libraries in linear algebra operations, such as solving linear least square problems, linear system of equations, or eigenvalue problems.

1) GEMM: General Matrix-Matrix Multiplication is part of the Level 3 of the BLAS specification [15] and is considered the main building block in dense linear algebra because many other operations can be expressed in terms of several GEMM invocations [16]. It is defined as follows:

$$C = \alpha A * B + \beta C \tag{1}$$

where A, B and C are matrices and α and β are scalars.

2) GEMV: General Matrix-Vector Multiplication belongs to Level 2 of the BLAS specification and is defined as follows:

$$y = \alpha A * x + \beta y \tag{2}$$

where A is a matrix, x and y are vectors and α and β are scalars.

B. Sparse

The Sparse Matrix-Vector multiplication (SPMV) is essential in NLA as it is one of the most time-consuming stages in many applications. An example is the solution of linear systems of equations using Krylov subspace methods. These methods require the repeated use of the SPMVkernel, keeping the same matrix and changing the dense vector.

The importance of this kernel has motivated numerous efforts targetting its optimization and performance improvement. Accompanying this effort is the historical evolution of HPC platforms, which is why there are efficient implementations of SPMV for the most widespread hardware platforms.

The serial version of the SPMV is shown in Algorithm 1, where the sparse matrix A is stored in Compressed Sparse Row (CSR) format. The nonzero elements are stored in vector val, the column index of each element in the matrix A is stored in a vector *col_idx*, and *row_ptr* stores the index of the first element for each row in vector val. The nonzero elements within each row are arranged by column index.

Algorithm 1 Serially computed sparse matrix-vector multiplication (SPMV) with the sparse matrix A stored in CSR format.

```
Input: $row\_ptr, col\_idx, val, x$
Output: $y$
$y = 0$
for $i = 0$ to $n - 1$ do
    for $j = row\_ptr[i]$ to $row\_ptr[i+1] - 1$ do
        $y[i] = y[i] + val[j] \cdot x[col\_idx[j]]$
    end for
end for
```

The main approach to parallelize this operation using the CSR format is to exploit the absence of data dependencies between the computations associated with each row. However, this approach presents severe load imbalances –depending on the sparsity pattern– and suffers from indirect data access to the dense vector x.

III. Evaluated kernels

In this section we introduce the Vitis Libraries and give an overview of the kernels chosen for our experiments. A review of the state-of-the-art in the use of FPGAs to compute dense and sparse NLA operations can be found in F. Favaro et al. [17], [18].

A. Vitis libraries

Xilinx's Vitis software includes a range of performance-optimized open source libraries for various purposes, including math, linear algebra, and computer vision. The Vitis BLAS Library, specifically, is an FPGA implementation of BLAS, which can provide significant performance benefits for applications that require intensive linear algebra operations.

The library offers three levels of implementation: primitives (L1), kernels (L2), and software APIs (L3). L1 includes parametrized C++ implementations of the basic operations found in BLAS, which can be compiled with HLS. These primitives consist of modules for computation and data movement, allowing the programmer to build high-performance logic by connecting computation and data mover modules. L2 provides kernel implementation examples for host code developers, and L3 offers C/C++ and Python APIs to accelerate BLAS operations using pre-built FPGA images.

The Vitis SPARSE library is a fast and efficient HLS implementation of the basic linear algebra subroutines for handling sparse matrices on FPGAs. The library includes two types of implementations: primitives (L1) and kernels (L2). The L1 primitives can be used by FPGA hardware developers to develop their own hardware designs, while the L2 kernels implementation provides usage examples for system and host code developers.

In this work, we evaluated the BLAS kernels from L2, which are divided in two groups. The kernels of the first one have the same top function, which has only two ports for communication with external memory (DRAM or HBM), and consist of an instruction processing block, a computation unit (e.g. GEMM), and a timer unit. The second group makes efficient use of the multichannel HBM memory to improve communication bandwidth. From sparse library we tested SPMV streaming kernel from L2.

1) GEMM basic: The architecture of this kernel from the first group is composed of the following blocks:

- Systolic array: Implemented using L1 primitives. Its size depends on the datatype and the memory interface. For single precision floating point and 512 bits interface it corresponds to 16×16.
- Data movers: These blocks get data from global memory and send it to the computation blocks, and vice versa.
- Transpose modules: One of the matrices must be transposed before entering the systolic array. This block also acts as a buffer to reuse data.

2) GEMM Multiple Compute Units (MCU): This kernel is implemented as two parallel instances (compute units) of the previous kernel. Each compute unit has its own dedicated HBM channel. The provided version of this kernel uses four compute units and its intended for the Alveo U250 board. In order to fit the design in the Alveo U50 board only 2 instances could be used. Also the DDR memory had to be changed for HBM.

3) GEMV basic: This kernel follows the same structure as GEMM basic, but with a custom processing block to perform GEMV operation.

4) GEMV streaming: This kernel is from the second group. To maximize throughput, it instantiates N parallel GEMV compute blocks primitives and connects each one to an individual HBM channel via data mover modules. N can be any number from 1 to 32, which is the maximum number of HBM channels in both Alveo boards. We evaluated this kernel with $N = 16$ for the Alveo U50 and with $N = 32$ for the Alveo U280.

5) Vitis SPMV: The SPMV accelerator consists of a group of compute units (CUs) connected via AXI STREAMs. In this design, 16 HBM channels are used to store the non-zero values and indices of a sparse matrix. Each HBM channel feeds a dedicated computation path to perform the SPMV operation for the portion of the sparse matrix data stored in that channel. As a result, 16 SPMV operations are performed simultaneously on different sections of the sparse matrix data. The partitioning of the sparse matrix data is done by the host.

B. Matrix-matrix multiplication (MMM)

To provide a point of comparison for the results of Vitis BLAS, we included in our evaluation a state-of-the-art implementation for the GEMM function developed by J. de Fine Licht et al. [19]. The authors propose an implementation of matrix-matrix multiplication (MMM) on an FPGA that aims to minimize off-chip data movement and maximize performance. They accomplish this by utilizing on-chip memory and by carefully designing the hardware architecture to be highly optimized for the resources available. The authors start by developing a general model for computation, I/O, and resource utilization in order to create a hardware architecture that is tailored to the specific capabilities of the target device.

The proposed implementation follows a systolic array architecture, in which N_p processing elements (PEs) consume pre-fetched elements of the matrices A and B in a stream-like manner. Each PE holds N_c compute units (CUs), and each CU is capable of producing one output product (a partial result of matrix C) every clock cycle. The degree of parallelism is determined by the number of CUs. The implementation is written in HLS C++ and is parametrized, portable, scalable, and open-source, which are rare characteristics for highly-tuned FPGA implementations.

C. HiSparse SPMV

Yixiao Du et al. [20] developed HiSparse, a high-performance kernel for HBM-equipped FPGAs. HiSparse uses a custom format for storing sparse matrices in HBM, allowing vectorized-streaming access to each HBM channel and concurrent access to multiple channels. This helps to fully utilize the available bandwidth of HBM for loading the sparse matrix. In addition, HiSparse features a scalable on-chip buffer design that incorporates vector replication and banking to support a large number of parallel processing engines (PEs). This helps to maximize data reuse in accessing the input vector. HiSparse is able to support arbitrarily large matrices through the use of matrix partitioning along both rows and columns.

IV. EXPERIMENTAL EVALUATION

A. Setup

We used the following hardware for the experiments:

- An Alveo U50 FPGA accelerator card from Xilinx. The FPGA is based on the UltraScale+ architecture and includes 872K look-up tables, 1743K registers, 28 MB of internal RAM, and 5952 DSP blocks. The chip also has 8 GB of HBM RAM. To compile the designs we used Xilinx Vitis 2022.2.

- An Alveo U280 FPGA accelerator card from Xilinx. The FPGA is based on the UltraScale+ architecture and includes 1303K look-up tables, 2607K registers, 41 MB of internal RAM, and 9024 DSP blocks. The board includes 8 GB of HBM and 32 GB of DDR RAM. To compile the designs we used Xilinx Vitis 2022.2.
- A system with an Intel Xeon Silver 4208 CPU with 8-cores running at 2.1 GHz, and 80 GB of RAM. The CPU implementations make use of Intel MKL library, using all 8 cores with SMT disabled and AVX2 instructions. This device is capable of AVX512, but we experimentally determined that using this feature in multicore execution severely limits the operating frequency of the cores, which degrades the performance.

We performed the characterization of performance and energy consumption as follows:

- For the Alveo platforms, the boards include sensors for current, voltage, and temperature measurements while the kernel is running. The driver Xilinx Runtime (XRT) provides these values to the host.
- For the Intel Xeon processor, we used RAPL to obtain estimations of CPU and memory power consumption.
- All power measurements were automated using PMlib [21]. To obtain final results we averaged readings collected during 2 minutes of execution, with an equal warm-up time before measuring.
- Runtime measurements are also the average of several kernel excecutions.

B. Experimental results and discussion

In our experimental evaluation, we use the GEMM, GEMV, and SPMV implementations from the MKL library on a CPU as a baseline. The results presented in this section are the average of 10 independent runs, all using single precision floating point arithmetic.

The resource utilization of the implemented FPGA kernels is shown in Table I.

In the first experiment we evaluate the computational performance and energy efficiency reached by the different versions for the GEMV operation over square matrices of: 128, 256, 512, 1024, 2048, 4096 and 8192 columns. Specifically, Table II presents the GFLOPs achieved by all the evaluated variants.

Based on the experimental results for the GEMV kernel, the basic version is a non-competitive option when considering the FPGA variants. This implementation has very low levels of parallelism because it performs the dot product on vectors of 16 elements. Additionally, there is a carry-dependency issue in the computation loop which causes it to operate 4 times slower than intended. For small test cases, the MMM variant performs better than the Streaming variants, which have poor performance on small matrices. However, for dimensions larger than 1024, the results are reversed. In particular, the performance of the MMM variant is stagnant for matrices with 1024 or more columns, while the Streaming variants continue to improve even for the largest matrices. The Streaming variants provides 16 and 32 times more parallelism than the basic version and takes advantage of the HBM on the Alveo board. The streaming variant for the Alveo U280 has twice as much paralellism than the Alveo U50 version, but this is not directly reflected in the performance for small or medium matrices. However, when the matrix size increases (for example, for size 16384) the performance improvement is more noticeable. The CPU version offers the best peak performance for matrices with 1024 columns, but its performance degrades for larger test cases. This result is reasonable considering the effects of cache memory usage.

The energy study, similar to the performance evaluation, shows that the fastest GEMV version is generally the most energy-efficient option. However, in all cases, the Alveo U50 FPGA implementations require less power than the CPU counterpart. Additionally, the MMM version uses less power, on average, than the Streaming variant. Finally, it is worth noting that the FPGA version outperforms the CPU counterpart in terms of energy consumption for the three largest dimensions. The Alveo U280 doubles the power consumption of the U50, and since the performance ratio is less than 2, the latter ends up being more energy efficient. On the other hand, the consumption of the U280 is similar to that of the CPU.

The experimental results for the performance and energy efficiency of the GEMM kernels are shown in Tables IV and V. In this case, the CPU variant outperforms the FPGA counterparts for all matrix dimensions. When focusing on the FPGA versions, the basic variant has significantly lower performance than the other versions. For the basic and MCU GEMM kernels, performance peaks around dimensions 1024 and then begins to decline. The reason for this performance loss for larger sizes is not fully determined and requires further investigation. The MMM variant also peaks around size 1024 but maintains constant performance for larger sizes. The Alveo U280 MCU version has similar performance than the U50 counterpart, which was expected since both have the same number of compute units. Given that the U50 consumes less power, it is more energy efficient.

In contrast to the performance results from the GEMV experiment, for GEMM the U50 FPGA outperforms the CPU in energy-efficiency for six out of seven matrix dimensions. This highlights the energy efficiency of FPGA platforms, particularly in this context where the CPU is faster than other versions. None of the Vitis BLAS versions outperform the CPU in this case (except for the smallest matrix size). This is expected since the evaluated kernels for GEMM were designed for larger FPGA boards and are not optimized for the Alveo U50 platform (unlike the Streaming GEMV which was designed for this board).

The performance results for the SPMV kernels are shown in Table VI. For this experiment we evaluated 22 sparse matrices from the SuiteSparse Matrix Collection with different sizes and number of nonzero elements. In general, it can be seen that the FPGA versions obtain better performance in most cases (16 out of 22). Within the FPGA versions, the Vitis version wins in all but one case. Analysing the characteristics of the matrices, it can be observed that the CPU version obtains

TABLE I
Resource utilization in percentage of available resources for the implemented FPGA kernels.

		Utilization (%)				
Type	Available	GEMV basic	GEMV Streaming	GEMM basic	GEMM MCU	MMM
LUTs	870016	14.02	22.83	37.39	61.16	42.45
Registers	1740032	8.98	17.10	28.30	47.59	32.33
Block RAM	1344	16.22	16.07	18.34	22.84	53.27
DSPs	5940	0.29	9.87	20.94	41.82	46.13

TABLE II
Achieved performance (GFLOPs) of the GEMV kernels for different matrix sizes.

		U50		U280	Xeon
Size	basic	Stream. 16ch	MMM	Stream. 32ch	MKL
128	0.20	0.43	1.21	0.35	5.74
256	0.43	1.48	4.48	1.30	19.75
512	0.60	5.86	7.15	5.71	42.35
1024	0.67	16.68	8.25	15.05	71.27
2048	0.63	29.59	9.15	36.68	44.24
4096	0.41	33.91	9.44	47.23	26.48
8192	-	39.34	9.54	59.37	23.42
16384	-	41.81	-	62.73	-

TABLE III
Energy efficiency (GFLOPs/W) of the GEMV kernels for different matrix sizes.

		U50		U280	Xeon
Size	basic	Stream. 16ch	MMM	Stream. 32ch	MKL
128	0.01	0.02	0.06	0.01	0.12
256	0.03	0.08	0.21	0.03	0.40
512	0.04	0.29	0.34	0.14	0.82
1024	0.04	0.77	0.38	0.35	1.36
2048	0.04	1.20	0.43	0.75	0.70
4096	0.03	1.31	0.44	0.90	0.41
8192	-	1.43	0.48	1.10	0.37
16384	-	1.47	-	1.15	-

TABLE IV
Achieved performance (GFLOPs) of the GEMM kernels for different matrix sizes.

		U50		U280	Xeon
Size	basic	MCU	MMM	MCU	MKL
128	18.79	170.90	5.14	187.20	220.51
256	41.29	202.60	38.05	215.59	370.73
512	78.89	227.64	184.57	172.19	386.07
1024	120.86	234.36	239.54	182.20	396.87
2048	115.39	229.61	261.11	187.60	424.29
4096	67.22	137.44	266.92	190.15	429.64
8192	-	139.13	269.23	188.23	369.73

TABLE V
Achieved energy-efficiency (GFLOPs/W) of the GEMM kernels for different matrix sizes.

		U50		U280	Xeon
Size	basic	MCU	MMM	MCU	MKL
128	0.97	6.90	0.21	3.40	3.87
256	1.95	5.78	1.46	3.46	6.32
512	2.84	5.55	8.21	3.77	6.57
1024	3.83	5.44	10.21	3.79	6.60
2048	4.04	5.56	10.99	3.81	6.65
4096	2.85	4.39	11.21	2.30	6.46
8192	-	4.54	11.27	2.38	6.00

better performance for the sparse matrices with a lower level of sparsity ($NNZ/(N \times M)$).

V. Conclusions

In this article, we extended our previous research [13] about the use of non-traditional HPC hardware for computing NLA kernels. We have reviewed the available kernels for computing the GEMV, GEMM and SpMV kernels on FPGAs and extended and tuned some variants of these kernels. The experimental evaluation carried out on two different Alveo FPGA boards shows that, in general, the CPU version outperforms the FPGA counterparts in terms of GFLOPs for the dense case, but the use of FPGAs offers more efficient variants in terms of energy consumption. These results are significant. First, because of the importance of energy consumption as a constraint in the HPC field, and second, because of the long history of CPU implementations compared to the recent focus on FPGAs for this kind of computation. For the sparse case, most of the matrices achieve better performance in the FPGA than the CPU. This predicts that from the point of view of energy consumption the benefits will be even greater.

There are several directions for future work. First, it is necessary to extend the study to include other FPGAs with different characteristics, including other Intel FPGAs. Second, the comparison should be expanded to include other heterogeneous hardware platforms, such as cutting-edge GPUs and low-power devices like ARM processors. Finally, it would be interesting to complement the GFLOPs and GFLOPs per watt metrics with other perspectives, such as the learning curve for FPGA design and the design and compilation times for FPGA implementations.

TABLE VI
ACHIEVED PERFORMANCE (GFLOPS) OF THE SPMV KERNELS FOR DIFFERENT SPARSE MATRICES.

Matrix	ROWS (N)	COLS (M)	NNZ	Vitis SpMV	HiSparse	CPU MKL
bcsstk15	3948	3948	117816	5.50	1.46	4.46
bcsstk24	3562	3562	159910	8.27	2.20	5.17
bcsstk28	4410	4410	219024	9.53	3.13	5.94
bcsstk36	23052	23052	1143140	16.05	9.30	13.18
bodyy4	17546	17546	121550	2.75	1.42	3.03
bodyy6	19366	19366	134208	2.78	1.62	3.21
cbuckle	13681	13681	676515	17.10	6.65	9.46
ct20stif	52329	52329	2600295	12.90	13.30	13.35
ex9	3363	3363	99471	5.37	1.41	4.78
gyro_k	17361	17361	1021159	12.79	8.78	12.88
msc10848	10848	10848	1229776	14.84	10.28	11.76
msc23052	23052	23052	1142686	7.68	8.87	11.10
nasa2910	2910	2910	174296	8.54	2.57	5.34
nasasrb	54870	54870	2677324	16.76	13.11	17.74
nd3k	9000	9000	3279690	22.98	18.04	19.11
nd6k	18000	18000	6897316	21.75	23.06	17.51
olafu	16146	16146	1015156	19.92	8.63	11.61
raefsky4	19779	19779	1316789	15.04	9.88	14.86
s2rmq4m1	5489	5489	263351	11.23	3.34	6.49
s3rmt3m3	5357	5357	207123	9.52	2.78	4.79
ted_B	10605	10605	144579	4.68	1.94	2.74
ted_B_unscaled	10605	10605	144579	4.68	1.92	3.00

ACKNOWLEDGMENT

We acknowledge ANII–MPG Indep.–Research–Group: “Efficient Hetergenous Computing” with the CSC group.

REFERENCES

[1] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. Siam, 1994, vol. 43.

[2] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006.

[3] J. J. Dongarra, P. Luszczek, and A. Petitet, “The linpack benchmark: Past, present, and future,” 2002.

[4] “The top500 list,” 2022, available at http://www.top500.org.

[5] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' guide*, 3rd ed. SIAM, 1999.

[6] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for Fortran usage,” vol. 5, no. 3, pp. 308–323, September 1979.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, “An extended set of FORTRAN basic linear algebra subprograms,” vol. 14, no. 1, pp. 1–17, March 1988.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, “A set of level 3 basic linear algebra subprograms,” vol. 16, no. 1, pp. 1–17, March 1990.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. SIAM, 1997.

[10] J. Dongarra *et al*, “The international ExaScale software project roadmap,” *Int. J. of High Performance Computing & Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[11] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón, “On the impact of optimization on the time-power-energy balance of dense linear algebra factorizations,” in *Algorithms and Architectures for Parallel Processing - 13th International Conference, ICA3PP 2013, Vietri sul Mare, Italy, December 18-20, 2013, Proceedings, Part II*, ser. Lecture Notes in Computer Science, R. Aversa, J. Kolodziej, J. Zhang, F. Amato, and G. Fortino, Eds., vol. 8286. Springer, 2013, pp. 3–10. [Online]. Available: https://doi.org/10.1007/978-3-319-03889-6_1

[12] P. Ezzatti, E. S. Quintana-Ortí, A. Remón, and J. Saak, “Power-aware computing,” *Concurrency and Computation: Practice and Experience*, vol. 31, no. 6, p. e5034, 2019, e5034 cpe.5034.

[13] F. Favaro, E. Dufrechou, J. P. Oliver, and P. Ezzatti, “Time-power-energy balance of blas kernels in modern fpgas,” in *High Performance Computing*, P. Navaux, C. J. Barrios H., C. Osthoff, and G. Guerrero, Eds. Cham: Springer International Publishing, 2022, pp. 78–89.

[14] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, “An updated set of basic linear algebra subprograms (BLAS),” *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[15] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, “A set of level 3 basic linear algebra subprograms,” *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, March 1990.

[16] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, T. Rhodes, R. A. van de Geijn, and F. G. Van Zee, “Deriving dense linear algebra libraries,” *Formal Aspects of Computing*, vol. 25, no. 6, pp. 933–945, Nov 2013.

[17] F. Favaro, J. P. Oliver, E. Dufrechou, and P. Ezzatti, “Understanding the Performance of Elementary NLA Kernels in FPGAs,” in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 479–482.

[18] F. Favaro, E. Dufrechou, P. Ezzatti, and J. P. Oliver, “Energy-efficient algebra kernels in FPGA for High Performance Computing,” vol. 21, Oct 2021.

[19] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, “Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis,” *In Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20), February 23-25, 2020, Seaside, CA, USA*, December 2020.

[20] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, “High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv,” in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 54–64. [Online]. Available: https://doi.org/10.1145/3490422.3502368

[21] S. Barrachina, M. Barreda, S. Catalán, M. F. Dolz, G. Fabregat, R. Mayo, and E. Quintana-Ortí, “An integrated framework for power-performance analysis of parallel scientific workloads,” *Energy*, pp. 114–119, 2013.

High-Speed Textural Image Features Extraction using FPGA

Jeremías Gaia*, Emanuel Trabes‡, Eugenio Orosco*, Francisco Rossomando* and Carlos Soria*
* Instituto de Automática, Facultad de Ingeniería, Universidad Nacional de San Juan, Argentina
‡ Departamento de Electronica, Universidad Nacional de San Luis, Argentina

***Abstract*—Computer vision algorithms are present in almost all robotics applications. High-level image operations like image classification, object tracking and pattern recognition need low-level information about the image texture to work. However, in order to include this compute-intensive tasks in a control loop, they must be executed in the shortest time possible. This article describes a high-speed embedded system to compute different texture-related image features. The system was tested for different image sizes in two Xilinx FPGA platforms: an Alveo U200 Data Center Accelerator Card and a Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. Results are evaluated in comparison to a desktop PC's performance.**

***Index Terms*—FPGA, Texture, Haralcik Features, Image Processing**

I. Introduction

Field Programmable Gate Arrays (FPGAs) can generate dedicated hardware with low latency and high throughput computation; these characteristics have made such devices a suitable choice for performing compute-intensive video and image processing. Some of the most common operations performed over images include color to gray conversion [1], histogram construction [2] and image filtering [3].

Texture information is a key component for detecting objects or regions of interest in an image. For example: in medical applications, texture information improves diagnostic imaging [4], [5]. This is due to evidence demonstrating that AI-based image classification algorithms can effectively exploit texture for pattern recognition [6], [7].

On the other hand, real-time applications like simultaneous localization and mapping (SLAM) [8], autonomous driving safety [9], or human-robot interaction [10] need this extraction process to be as fast as possible in order to perform as expected.

Another aspect of image processing connected to texture analysis are gradient images. By applying directional filters to the original image, edges can be easily extracted, thus allowing the system to detect objects or patterns [11].

On the other hand, contrast has a significant visual impact on an image, since it highlights or hides textures. The shape of the image histogram can be used to examine this element [12], which makes it a useful support measure for evaluating texture.

Similar to our work, in [13] an embedded architecture for the GLCM calculation was presented. The authors proposed a hardware design able to construct GLCM matrices from image patches of 128x128 pixels.

In this paper, an FPGA implementation to calculate a set of statistical image features for texture analysis is described. The input image is processed in order to obtain histogram and gradient-based image features as well as the Haralick texture features [14]. Computations are performance-optimized by exploiting FPGAs capacity to integrate sequential and parallel processing.

The rest of the paper is organized as follows: In Sec. II, the expressions used to determine different statistical image features are presented. Details about how the system is implemented are shown in Sec. III. Sec. IV shows Experimentation results. Finally, Sec. V concludes the article.

II. Theoretical Background

A. Histogram-based measures

In this article, three histogram-based metrics were used: *histogram flatness measure (HFM), histogram spread (HS), and global entropy.*

1) Histogram Flatness and Histogram Spread: Introduced by Tripathi *et. al.* in [12], these metrics are very useful to assess image contrast. HFM can be expressed as the ratio of the geometric mean of the image histogram $h(x)$ to the arithmetic mean of $h(x)$ (Eq. (1)).

$$HFM = \frac{\left\{\prod_{i=1}^{n} x_i\right\}^{\frac{1}{n}}}{\frac{1}{n}\sum_{i=1}^{n} x_i} \tag{1}$$

where x_i is the count for the i-*th* histogram bin and n the number of histogram bins.

Histogram spread is the ratio of the inter-quartile distance to the range of the histogram. Here, the inter-quartile distance is the difference between the 3*rd* and the 1*st* quartile of the cumulative histogram.

$$HS = \frac{3^{rd}quartile - 1^{st}quartile}{max(h(x)) - min(h(x))} \tag{2}$$

2) Image Entropy: The global entropy of an image (Eq. (3)) measures the amount of low-level information contained in it. The probability for a pixel taking certain value in an image can be represented as the bin count for the given gray tone in the image histogram.

$$entropy = \sum_{i=0}^{255} h(i)\log_2(h(i)) \tag{3}$$

B. Gradient-based measures

1) Gradient Magnitude: An initial approach that could be used to measure image sharpness is to consider the mean value of a *magnitude image*. Being G_x the gradient image in the x direction and G_y the gradient image in the y direction, the magnitude image (G) can be defined as $G = \sqrt{(G_x)^2 + (G_y)^2}$

Then, the mean value of G is

$$GradMagnitude = \frac{1}{rows * cols} \sum_{i=1}^{rows} \sum_{j=1}^{cols} G(i,j) \tag{4}$$

Greater values of the gradient magnitude measure indicate the presence of greater textural information in the image.

2) γ Sharpness: Shin *et.al.* [15] proposed to apply the mapping function in Eq. (5) to the magnitude image G.

$$\hat{g}_i = \begin{cases} \frac{1}{N_g} log(\lambda(g_i - \gamma) + 1), & \text{for } g_i \geq \gamma, \\ 0, & \text{for } g_i < \gamma, \end{cases} \tag{5}$$

where $N_g = log(\lambda(1 - \gamma) + 1)$ is the normalization factor, g_i denotes the gradient magnitude at pixel i, γ indicates the activation threshold value for the mapping function, λ is a control parameter to adjust the mapping behaviour and $\hat{g}_i$ stands for the amount of gradient information at pixel i. Note that $g_i \in [0, 1]$, assuming a pixel range of $[0, 255]$ and a normalization factor of $1/255$. After g_i is calculated the γ sharpness measure can be estimated as $\gamma_{sharp} = \sum \hat{g}_i$

C. Haralick Textural Features

In 1973, Robert Haralick proposed a new approach to extract textural information from an image: the graylevel co-ocurrence matrix (GLCM) [14]. According to this method, the spatial relationship between the gray tones in an image I encodes the texture information for that image.

The first step to extract this information is to create the GLCM. After that, a set of features can be obtained from this matrix. This work considers five textural features: *contrast, homogeneity, entropy and the information measures of correlation.*

In order to further understand the following equations, notation is provided.

- P refers to the graylevel co-ocurrence matrix.
- $p(i, j)$ is the (i, j)th entry in the normalized graylevel co-ocurrence matrix.

  $$p(i, j) = P(i, j)/R, \tag{6}$$

  being R a normalization factor.
- $p_x(i)$ refers to the ith entry in the marginal-probability matrix obtained by summing the rows of the co-ocurrence matrix P.

1) Angular Second Moment (energy):

$$f_1 = \sum_i^{N_g} \sum_j^{N_g} \{p(i, j)\}^2 \tag{7}$$

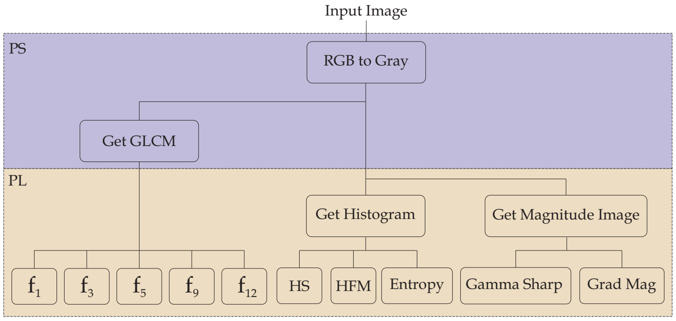


Fig. 1. Overview of the proposed system based on the Zynq UltraScale+ MPSoC ZCU104 architecture.

2) Contrast:

$$f_3 = \sum_i^{N_g-1} n^2 \left\{ \sum_{|i-j|=n} \sum_i^{N_g} \sum_j^{N_g} p(i, j) \right\} \tag{8}$$

3) Homogeneity:

$$f_5 = \sum_i^{N_g} \sum_j^{N_g} \frac{1}{1 + (i - j)^2} p(i, j) \tag{9}$$

4) Entropy:

$$f_9 = -\sum_i^{N_g} \sum_j^{N_g} p(i, j) \log(p(i, j)) \tag{10}$$

5) Information Measures of Correlation (IMC):

$$f_{12} = \frac{HXY - HXY1}{max\{HX, HY\}} \tag{11}$$

with HXY being the same as f_9. HX and HY are entropies of p_x and p_y respectively, calculated with expression (10). Then

$$HXY1 = -\sum_i^{N_g} \sum_j^{N_g} p(i, j) \log\{p_x(i)p_y(j)\} \tag{12}$$

Correlation metrics report the presence of linear dependencies in the image. In the case of the IMC measure, by combining different correlations it provides a reference of the amount of organized structure in the image.

III. Implementation details

Modern FPGAs' can be considered a unit of two main blocks: a Processing System (PS) block and a Programmable Logic (PL) Block. The first normally contains an embedded processor, while the second contains designed hardware [16]. Both blocks are carefully employed in this article, with the PL being used for the majority of the calculations and the PS being used for operations that cannot be parallelized.

Recently, Xilinx company has developed the xfOpenCV library [17]. A set of performance-optimized kernels for Xilinx FPGAs' and SoCs, based on the OpenCV computer vision library [18] are made available for users. It allows to translate

TABLE I
COMPARISON WITH STATE-OF-THE-ART RESEARCH.

Task	Article	Image Size	Time[ms]	Platform
Haralick features extraction	Sieler *et.al.* [13]	512x512	101.50	Virtex-XCV2000
	Sieler *et.al.* [13]	512x512	37.00	Virtex-XC5VLX50T
	Ours	640x480	2.40	ZCU104
	Ours	640x480	1.09	AlveoU200
Histogram Computation	Younis *et.al.* [2]	640x360	1.15	Virtex XC4VSX35
	Ours	640x480	2.40	ZCU104
	Ours	640x480	1.09	AlveoU200
Magnitude image estimation	Tsiktsiris *et.al.* [3]	640x480	98.00	Altera Cyclone IV EP4CE11
	Ours	640x480	2.40	ZCU104
	Ours	640x480	1.09	AlveoU200

some of the most common computer vision operations from sequential software processing to parallel hardware processing.

Register-transfer-level abstraction (RTL) is used in hardware description languages to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived [19]. By using the xfOpenCV kernels the programmer is relieved of the internal connections of the system, since the library automatically generates the RTL code.

Figure 1 shows an overview of the system. As a first step, the processing system (running a PetaLinux [20] distribution) reads an image stored in memory. Then, it is converted from RGB to gray by using the traditional OpenCV computer vision library.

The resulting grayscale image is fed into a function that performs the GLCM computation. Given the sequential nature of the GLCM constrction process, it is necessary for this operation to live in the PS. The GLCM and the grayscale image are sent to the programmable logic via write buffers.

For the design to be as fast as possible, the dataflow paradigm was adopted. The `DATAFLOW` directive enables task-level pipelining, allowing functions and loops to overlap in their operation. This is a key factor for parallel computation, decreasing latency and improving the throughput of the generated RTL.

Once the data is transferred to the programmable logic portion of the FPGA, the xfOpenCV library comes into play. The data exchange format between functions in the PL is an adaptation of OpenCV's `cv::Mat` named `xf::cv::Mat`.

The grayscale image is streamed both to the histogram and magnitude image generation blocks. The former calculates the histogram and its associated features, while the latter computes the gradient image and its related metrics. Simultaneously, the GLCM-dependent textural features are computed.

Once the feature extraction process is done, results are written into a buffer to return to the PS. Since this stage of the research is focused on the feature extraction process, results are printed to the screen. Future work may include this features in a control loop.

The source code files were developed in the Xilinx Vitis IDE v2021.2 and Vitis HLS IDE v2021.2. The xfOpenCV library V2022.1 was installed along with the OpenCV V4.4.0 library. GCC 7, G++ 7 and CMake 3.16 were used for compilation.

TABLE II
LATENCY [MS] FOR DIFFERENT IMPLEMENTATIONS OF OUR SISTEM.

Platform	Image Size					
	128x128	320x240	400x300	512x384	640x480	720x480
AlveoU200	0.25	0.31	0.4	0.72	1.09	1.22
ZCU104	0.74	0.85	1.14	1.66	2.40	2.76
PC	104.06	159.11	148.34	230.32	191.42	194.08

IV. RESULTS

Our design was simulated, synthesized and tested in two separate platforms: a Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit and an Alveo U200 Data Center Accelerator Card. Also, a software version of the system was evaluated for comparison purposes using a computer with an Intel Core i9-10900 CPU running at 2.8 GHz with 16GB of RAM and Ubuntu 20.04 LTS operating system.

Most state-of-the-art research focuses on accelerating only one image operation: performing RGB to gray conversion, histogram calculations, Haralick features extraction, etc. On the other hand, our proposal is able to perform multiple computations at the same time and with a single pass of the image. Despite this difference, Table I shows a comparison between our system and related research for different tasks. Results shown in this table correspond to those declared by the authors in the original publication.

Sieler *et.al.* [13] implemented their system for Haralick features extraction, on two Virtex boards. For a 512x512 image the Virtex-V board presents better results. However, for a bigger image, our implementation outperforms [13]. For the histogram computation task, our system is compared to [2]. In this case, both the AlveoU200 implementation of our system and [2] itself have similar performance. However, for the ZCU104, the system needs two times more processing time.

The magnitude image estimation portion of Table I presents a clear advantage for our proposed system. Both implementations outperform the work by Tsiktsiris *et.al.*.

The highly parallel nature of the PL part of our system allows to perform multiple tasks in Table I at the same time. This explains the fact that the measured times for the different tasks is the same.

Table II compares the performance of both implementations of the proposed system versus a desktop PC. Latency times were measured feeding the system with different versions of an input image. The performance metric is the time difference between the start of the grayscale image and GLCM stream to the PL and the end of the last metric computation. The aforementioned time measurements were obtained through a combination of C-coded internal calculations on the deployed system and visual inspection of the ”live waveform viewer tool” of the Vivado Behavioral Simulation.

The reader may note that the calculation times between the two platforms tend to diverge as the image size grows. However, for the largest image size, latency time for the ZCU104 board is under 3 ms.

V. Conclusions

Low-latency algorithms for compute-intensive image processing were successfully implemented, providing the user with a set of common statistical image features that can be leveraged for different applications.

The system was tested for a set of input images in two Xilinx platforms with different hardware resources. A slight difference in favor of the one of more capabilities was detected.

The performance difference confirms that the design is not platform-independent in terms of calculation times. However, it is suitable for inclusion in a control loop, since they are able to finish calculations under 3 ms.

Acknowledgements

J. Gaia would particularly like to thank the High Performance Computing and Networking Research Group for their cooperation and the Secretaría de Ciencia, Tecnologa e Innovación (SECITI) of San Juan, Argentina, for their financial support.

Conflict of interests

Authors declare no conflict of interests.

References

[1] Y. Zhang, X. Yang, L. Wu, and J. H. Andrian, "A case study on approximate fpga design with an open-source image processing platform," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019.

[2] D. Younis and B. M. Younis, "Low cost histogram implementation for image processing using fpga," in *IOP Conference Series: Materials Science and Engineering*. IOP Publishing, 2020.

[3] D. Tsiktsiris, D. Ziouzios, and M. Dasygenis, "A portable image processing accelerator using fpga," in *2018 7th Int. Conf. on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 2018.

[4] S. Samanta, S. S. Ahmed, M. A.-M. M. Salem, S. S. Nath, N. Dey, and S. S. Chowdhury, "Haralick features based automated glaucoma classification using back propagation neural network," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014: Volume 1*. Springer, 2015, pp. 351–358.

[5] H. Soltanian-Zadeh, F. Rafiee-Rad *et al.*, "Comparison of multiwavelet, wavelet, haralick, and shape features for microcalcification classification in mammograms," *Pattern recognition*, vol. 37, no. 10, pp. 1973–1986, 2004.

[6] V. Bhateja, A. Gautam, A. Tiwari, L. N. Bao, S. C. Satapathy, N. G. Nhu, and D.-N. Le, "Haralick features-based classification of mammograms using svm," in *Information Systems Design and Intelligent Applications: Proceedings of Fourth International Conference INDIA 2017*. Springer, 2018, pp. 787–795.

[7] N. Zayed and H. A. Elnemr, "Statistical analysis of haralick texture features to discriminate lung abnormalities," *Journal of Biomedical Imaging*, vol. 2015, pp. 12–12, 2015.

[8] S. Aldegheri, N. Bombieri, D. D. Bloisi, and A. Farinelli, "Data flow orb-slam for real-time performance on embedded gpu boards," in *2019 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 2019.

[9] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, 2019.

[10] J. Webber, A. Mehbodniya, R. Teng, A. Arafa, and A. Alwakeel, "Finger-gesture recognition for visible light communication systems using machine learning," *Applied Sciences*, 2021.

[11] N. Kanopoulos, N. Vasanthavada, and R. L. Baker, "Design of an image edge detection filter using the sobel operator," *IEEE Journal of solid-state circuits*, vol. 23, no. 2, pp. 358–367, 1988.

[12] A. K. Tripathi, S. Mukhopadhyay, and A. K. Dhara, "Performance metrics for image contrast," in *2011 Int. Conf. on Image Information Processing*. IEEE, 2011.

[13] L. Siéler, C. Tanougast, and A. Bouridane, "A scalable and embedded fpga architecture for efficient computation of grey level co-occurrence matrices and haralick textures features," *Microprocessors and Microsystems*, 2010.

[14] R. M. Haralick, K. Shanmugam, and I. H. Dinstein, "Textural features for image classification," *IEEE Transactions on systems, man, and cybernetics*, 1973.

[15] U. Shin, J. Park, G. Shim, F. Rameau, and I. S. Kweon, "Camera exposure control for robust robot vision with noise-aware image quality assessment," in *2019 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 2019.

[16] Y. Nitta, S. Tamura, H. Yugen, and H. Takase, "Zytlebot: Fpga integrated development platform for ros based autonomous mobile robot," in *2019 Int. Conf. on Field-Programmable Technology (ICFPT)*. IEEE, 2019.

[17] "Xilinx opencv library," https://github.com/Xilinx/xfopencv, [Online; accessed 13-February-2023].

[18] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[19] F. Vahid, *Digital design with RTL design, VHDL, and Verilog*. John Wiley & Sons, 2010.

[20] "Petalinux tools," https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html, [Online; accessed 16-February-2023].

Multi-stage multirate filterbank for FPGA resource optimization

L. H. Arnaldi
Laboratorio Detección de Partículas y Radiación
Bariloche Atomic Center and Instituto Balseiro, National Commission of Atomic Energy
S. C. de Bariloche, RN, Argentina
arnaldi@cab.cnea.gov.ar

***Abstract*—Radio astronomy instrumentation is one of the scientific fields driving the design of more efficient filterbank architectures to manage the significant frequency channelisation task required to meet its scientific goals. FPGAs are the target devices for these tasks due to its flexibility and digital resource availavility. In this paper the problem of optimization of multirate filter banks is addressed. The factors that define the efficiency of these multirate systems are investigated and the implementations of structures in stages are analyzed. The latter, together with the polyphase implementations of the filters, allows obtaining optimal filterbanks in the use of resources for the FPGAs.**

***Index Terms*—Multi-rate, multi-stage, filterbank, optimization**

I. Introduction

One common task in telecommunications is to separate communication channels coming together in a wideband signal. This task result in a considerable workload for the electronics [1], [2]. A similar problem is found in the radio astronomy comunity, requiring precise meassurements to meet its scientific goals. It is required state-of-the-art sensitive detectors, extended control of instrumental systematic effects and accurate subtraction of foregrounds emitted by the sky. The current challenge in radio astronomy is to manufacture detector matrices for image studies in an extremely large format to achieve wide fields of view in the instruments and place image matrices in space. Associated to this is the challenge of achieving the manufacture of electronic excitation and readout systems capable of working together with these large detector arrays [3], [4].

Radio astronomy instrumentation is continually seeking greater processing bandwidths whilst maintaining output frequency resolutions on the order of some kHz per readout channel. With the recent commercial availability of multi Giga sample per second analog to digital converters (ADC), instantaneous processing bandwidths of the order of hundreds of MHz to many GHz are now viable for radio astronomy applications [5]. It is now possible to read large arrays of superconducting micro-resonators, such as the microwave kinetic inductance detectors (MKIDs), thanks to improvements in the cold and warm readout techniques [6]–[8].

To reduce the readout cost per sensor and the complexity of integration, efforts are currently focused on achieving higher multiplexing density, while keeping the readout noise below the intrinsic detector noise and presenting manageable thermal loads [4], [9].

In these large superconducting array readout systems, the bandwidth of the individual resonators is much less than the bandwidth of the multi-tone signal transmitted through the entire array. In this case, changing the sampling rate (decimating or interpolating) the signal in a single stage can result in a very expensive and computationally complex filtering arrangement. For these cases, it may be more efficient to implement the conversion of the sampling rate in multiple stages [10], [11]. The gradual decrease in the sampling rate results in a simplified design of the various filter stages and the resulting design can still meet the general decimation and filtering requirements. The savings are achieved by reducing the number of multiplications and additions per second (MADS), or what is the same, reducing the multiplication rate (MPS).

The theory developed in this paper applies to both interpolation and decimation systems, but the focus will be on decimation systems as this is the processing required in the analysis filterbanks used as readout systems in superconducting micro-resonator detectors. The application of these structures to a particular hardware platform is also investigated by mean of examples in order to test its performance in terms of number of multiplications, additions, and resource utilization. The design steps and some test results are shown for a Red Pitaya 125-14 board [12], targeting a SoC of the Zynq-7000 (xc7z010clg400-1) family. We focus on the development and implementation of efficient filterbanks to extract a set of channels that exist in a single sampled data stream. This data stream can be, for instance, the result of passing a multi-tone signal through a matrix of superconducting micro-resonators.

We also consider the design of oversampled discrete Fourier transform filterbanks using finite impulse response (FIR) filters in their polyphase form as decimators. This implementation is targeted to new field-programmable gate array (FPGA) technologies using standard language libraries and shows to be extremely efficient in the use of resources.

II. Digital filterbanks

Filterbanks are generally categorized as two types, *analysis* and *synthesis* filterbanks [13]–[15]. One is basically the inverse of the other. Fig. 1 show the structure of a K-channel analysis and synthesis filterbank. The $x(n)$ signal is the frequency

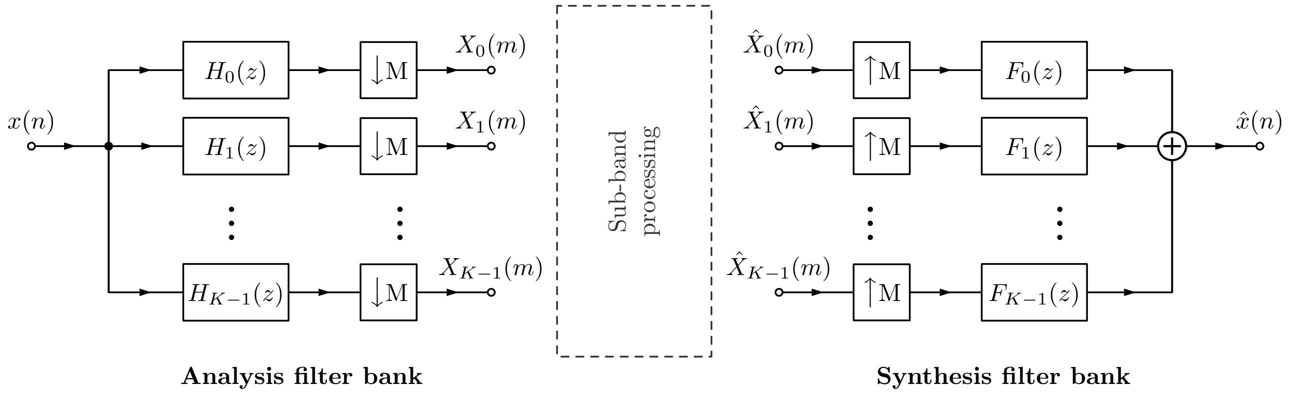


Fig. 1. Structure of a K-channel analysis and synthesis filterbank. The $x(n)$ signal is the multi-tone signal containing the channels of interest. The combination of delays, plus down-sampling by M ($\downarrow M$) can be seen as a commutator delivering one input sample to each of the filters ($H_k(z)$) at a time and at a low sampling rate (m). In the analysis filterbank, every output channel $X_k(m)$ is recovered at the output of a M-point DFT block. Here, the n and m ($n > m$) indexes denote a signal working at the input and output sampling rate, respectively. In the synthesis, the inverse process occurs.

division multiplexed (FDM) signal containing the channels of interest. The combination of delays, plus down-sampling by M ($\downarrow M$) delivers one input sample to each of the $H_k(z)$ filters at a time and at a low sampling rate (m). Here k is the channel number (or the sub-band number) and z is the z-transform variable. In the analysis filterbank, every output channel $X_k(m)$ is recovered at the output of a M-point DFT block. Here, the n and m (with $n > m$) indexes denote a signal working at the input and output sampling rate, respectively. In the synthesis, the inverse process occurs.

The approach in these systems is, in general, to find the perfect reconstruction (PR) condition [15], [16], where $x(n) = \hat{x}(n)$, with no more distortion other than a shift in time and some scale of amplitude.

An analysis filterbank consisting of K filters $h_k(n), \quad k = 0, 1, ..., K-1$ is called a *critically sampled* or *uniform* filterbank if $h_k(n)$ are derived from a prototype filter $h(n)$, where

$$h_k(n) = h(n)W_K^{-(k+k_0)n}, \quad k = 1, 2, ..., K-1, \tag{1}$$

with k_0, the frequency origin (normally $k_0 = 0$) and $W_K = e^{-j2\pi/K} = \sqrt[K]{1}$, the *twiddle factor*. Hence, the frequency response characteristics of the filters $h_k(n)$ are simply obtained by uniformly shifting the frequency response of the prototype filter in integer multiples of $2\pi/K$ [17]. In the uniform filterbank, $K = M$ or, equivalently $I = K/M = 1$, where M is the downsampling ratio of the filterbank and I is defined as the *oversampling ratio*. The frequency spectrum is then partitioned in a uniform manner. The sub-band width $\Delta_k = \frac{2\pi}{K} = \frac{F_s}{K}$ is identical for each sub-band and the band centers are uniformly spaced at intervals of $\frac{2\pi}{K} = \frac{F_s}{K}$, with a system working at a sampling rate F_s.

III. Changing the sampling rate in stages

The design of digital FIR filters with very narrow transition bands, flat passband, and large attenuation band results in high-order filters. The digital anti-aliasing and anti-image filters used in the filterbanks, critically sampled or oversampled, belong to this category. The focus now is on using a multistage design so that the design specifications of each individual filter can be relaxed.

A. Decimate in stages

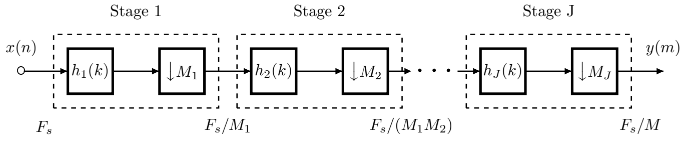


Fig. 2. Multi-stage decimation process.

Fig. 2 shows the arrangement of the blocks and the relationship between the frequencies for a decimation process in J stages. The condition for allowing multi-stage decimation is that M, the total decimation factor, is not a prime number and can be written as the product:

$$M = \prod_{i=1}^{J} M_i = M_1 \times M_2 \times \cdots \times M_J, \tag{2}$$

where $M_1, M_2, \ldots, M_J$, are the decimation factors associated with the respective stages in a J-stage decimator. This in turn implies that the original sample rate, F_s, and the final sample rate $F_{s,J}$ will be related according to

$$F_{s,J} = \frac{F_s}{M} = \frac{F_s}{\prod_{i=1}^{J} M_i}, \tag{3}$$

and the relationship between two successive sampling rates can be expressed as

$$F_{s,i} = \frac{F_{s,i-1}}{M_i}, \quad i = 1, 2, \ldots, J. \tag{4}$$

The initial and final frequencies will be $F_{s,0} = F_s$ and $F_{s,J} = F_s/M$, respectively.

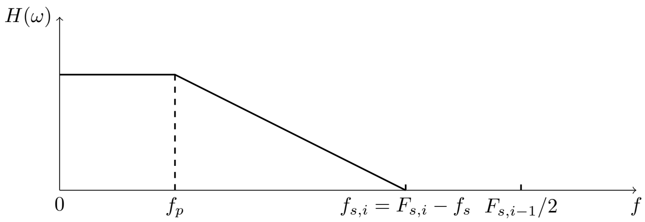


Fig. 3. Frequency response specifications for filter i, $i = 1, 2, \ldots, J$.

TABLE I
REQUIREMENTS FOR INDIVIDUAL STAGE FILTERS

Parameter	Value
Passband	$0 \leq f \leq f_p$
Stopband	$(F_{s,i} - \frac{F_s}{2M}) < f < \frac{F_{s,i-1}}{2},\ i = 1, 2, \ldots, J$
Passband ripple	δ_p / J
Stopband ripple	δ_s
Filter length	$N_i \simeq \frac{D_\infty(\delta_p, \delta_s)}{\Delta f_i} - f(\delta_p, \delta_s)\Delta f_i + 1$

B. Filter requirements for individual stages

The multi-stage decimation process can be analyzed from a spectral perspective. Fig. 3 and Table I present the characteristics required for the i-th stage of a multi-stage decimator filter like the one shown in Fig. 2 where $F_{s,i}$ and N_i are, respectively, the output sample rate and the length of the filter for the i-th decimator. The maximum frequency of the passband for the whole set is given by f_p and the start frequency of the stopband is f_s. The parameter Δf_i is the normalized transition band for the i-th stage and is given by

$$\Delta f_i = \frac{f_{s,i} - f_p}{F_{s,i-1}}. \tag{5}$$

In general, to avoid any degradation due to aliasing, the start frequency of the last stage's stopband must obey the sampling theorem, i.e.

$$f_s \leq \frac{F_{s,J}}{2}. \tag{6}$$

The characteristics of the passband of the filter of the i-th stage must be flat between zero frequency and f_p to avoid any distortion of the signal in that band. On the other hand, to ensure that there are no aliasing components in the desired signal band, the start frequency of the stopband of the i-th stage filter must be set to the relation:

$$f_{s,i} \leq F_{s,i} - f_s. \tag{7}$$

The result in (7) implies that the starting frequency of the stopband of the last stage is $f_{s,J} = F_{s,J} - f_s = 2f_s - f_s = f_s$, as required for the whole system.

At this point, the next parameters to analyze are the ripples allowed in the pass and attenuated bands of each stage. The requirement is that the overall passband ripple of the J-stage composite filter be below $1 + \delta_p$. This places more stringent restrictions on the tolerable passband ripple in the individual cascaded stages. A simple, but not unique, way of specifying the passband of each stage is to require that $1 + \delta_{p,i}$, be specified such that $\delta_{p,i} = \delta_p / J$. Similarly, the stopband ripple δ_s specifies the overall stopband ripple, or the stopband ripple of the cumulative multi-stage filter. This ripple is imposed on all cascaded stages, that is, $\delta_{s,i} \cong \delta_s$. This is particularly important since aliasing is not necessarily added consistently, so sufficient damping is required to avoid aliasing at each stage. However, it is up to the designer and the nature of the design to specify this parameter.

Having understood the relationship between the various parameters relating the responses of the different stages in a multi-stage decimator, an interesting question now arises: what is the optimal number of decimation stages and what is the amount by which to decimate at each stage to achieve maximum efficiency?. The answer to these questions lies in minimizing the computational complexity of the overall filter, which in turn is related to the expression for the length of the filter (N) that appears in Table I [10], [18].

The ultimate goal of a multi-stage decimator is to minimize the total number of multiplications and additions per second (MADS), R_T. Let R_i be the number of MADS in the i-th stage of decimation, then R_T is given by

$$R_T = \sum_{i=1}^{J} R_i. \tag{8}$$

Minimizing R_T minimizes the total amount of computation required in the system. Here it should be noted that R_T is nothing more than another way of expressing the MPS for the global system. Then, given the optimal filter length $N_{i,opt}$ for the i-th stage and assuming a direct form implementation for FIR filters, the parameter R_i can be expressed as

$$R_i = \frac{N_{i,opt} \times F_{s,i}}{M_i}. \tag{9}$$

From (4) and (9), in [11] it was found that (8) can be written as

$$R_T \cong D_\infty \left(\frac{\delta_p}{J}, \delta_s \right) F_s S \text{ (MADS)} \tag{10}$$

where

$$S = \frac{2}{\left(\Delta F \prod_{j=1}^{J-1} M_j \right)} + \sum_{i=1}^{J-1} \frac{M_i}{\left(\prod_{j=1}^{i} M_j \right) \left(1 - \alpha \prod_{j=1}^{i} M_j \right)}, \tag{11}$$

which has the final form, $\alpha = \frac{2 - \Delta F}{2M}$ and $\Delta F = \frac{f_s - f_p}{f_s}$. D_∞ is a parameter that depends on δ_p and δ_s. Its development is out of the scope of this paper, but the interested reader can be found it on [11].

Similarly, the theory can be developed to find the total memory cost of the system, N_T, defined as

$$N_T = \sum_{i=1}^{J} N_i, \tag{12}$$

where N_i is the number of coefficients of the filter of the i-th stage. This cost function can be expressed in terms of $D_\infty(\delta_p, \delta_s)$ as

$$N_T = D_\infty(\delta_p, \delta_s)GT, \tag{13}$$

where G is a constant of proportionality related to the implementation of the filter coefficients and T is given by

$$T = \frac{2}{\Delta F}\frac{M}{\prod_{j=1}^{J-1} M_j} + \sum_{i=1}^{J-1} \frac{M_i}{1 - \alpha \prod_{j=1}^{i} M_j}. \tag{14}$$

In other words, the problem of minimizing R_T or N_T is related to minimizing the functions S in (11) and T in (14), respectively.

IV. Optimal number of stages required

The design of multi-stage filters is a complex multidimensional optimization problem, although in [19] and [20] it was found that optimal solutions can be derived analytically by taking the partial differential equation of the cost function, reducing it to a one-dimensional problem without the need for complex numerical search algorithms. However, the optimal solutions are often groups of non-integer real numbers that cannot be implemented in practical systems. Manual adjustment of the results is needed, still requiring numerical methods to solve the equations, and for each design, the roots of the equation must be put back into a cost function to find the optimal solution set. In [21] this problem is represented in the integer domain using the grouping theory, and then performs the integer factorization. In [22] it is shown that the problem can be solved by an exhaustive search using genetic algorithms. In general, approaches that produce useful integer results have a high computational cost and do not consider important design properties of multi-stage filters [23].

Here we develop a simplified algorithm to directly search for optimal integer groups. Given the most useful practical design parameters, optimal results can be approximated with a limited number of assemblies for any design that satisfies certain constraints, with negligible cost. This greatly simplifies the complexity of the problem.

A. Search algorithm

The choice of the number of stages, J, and of the decimation factors, M_i, is not a trivial problem. However, in practice the number of stages rarely exceeds 3 or 4 [24]. Furthermore, for a given value of M, there is a limited set of possible integer factors. Then a possible solution to the problem of finding optimal decimation factors is to determine all possible factors of M, that is, all possible sets of M_i values and their corresponding requirements for R_T or N_T. The most efficient (or preferred) solution will then be chosen by inspection.

However, looking at the results of the above mentioned real-value and integer-value optimal solution experiments, there are two important properties of optimal solution distributions for computational cost and storage memory cost:

1) the set of M_i is always presented in descending order for multi-stage decimation and in ascending order for multi-stage interpolation [11], [24].

2) ΔF is related to the width of the transition band. Varying ΔF changes the order of the filter but not the sample rate shift factor (M_i) of each stage.

Therefore, the search for integer-valued solutions can be informed by 1), and because of 2), the size of the problem is considerably smaller than it appears to be.

Based on these properties, we developed an algorithm to search for the number of optimal stages that would allow us, given a certain design requirement, to obtain the most appropriate implementation in terms of the use of resources for the FPGA that would house the processing system. The algorithm is written in Python language and allows to obtain all the useful parameters of each stage of the studied system.

V. Evaluation of implementations using the search algorithm

To verify the algorithm developed, different designs of the filterbank with $K = 16$, critically sampled and oversampled channels, single-stage and multi-stage, were analyzed and implemented, before going on to analyze the characteristics of larger designs with $K \geq 10000$ channels. In each case, the optimal implementation for multi-stage designs was sought. The designs were tested on the Red Pitaya board.

A. Critically sampled filterbank $K = M = 16$ and oversampled filterbank $K = MI = 16$

The general specifications of the filter are the following: a decimator is designed with $M = 16$, which converts the frequency $F_{s,0} = 125\,\text{MHz}$ to the output sample rate of $F_{s,J} = 7.8125\,\text{MHz}$. The passband edge frequency is $f_p = 3.125\,\text{MHz}$ and the stopband edge frequency is $f_s = 3.90625\,\text{MHz}$. The maximum requested passband ripple is specified at $\delta_p = 5.756 \times 10^{-3}$ (0.1 dB), and the maximum stopband ripple is $\delta_s = 5.041 \times 10^{-4}$ (66 dB).

Applying the algorithm for finding the optimal number of stages, the decimation factor $M = 16$ can be expressed as a product of two integers, that is, $M = M_1 \times M_2$ and instead of single-stage decimation, this processing can be done in two steps. The algorithm finds two possible combinations, $M = 4 \times 4$ or $M = 8 \times 2$. After analyzing the results, it is found that the combination $M = 8 \times 2$ is the most efficient in terms of R_T. The choice of these factors yields $N_1 = 44$ and $N_2 = 61$ for the first and second stages, respectively. Fig. 4(a) shows the arrangement of the filtering and decimating blocks for the chosen configuration. In the first stage, the input signal

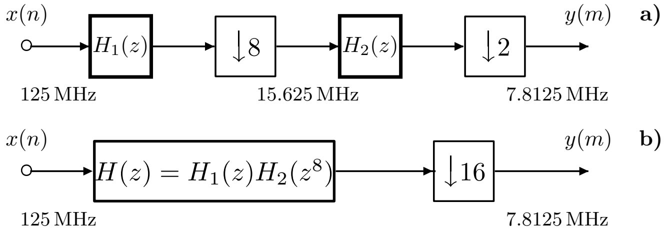


Fig. 4. Two stage decimator. a) Implementation for $M = 8 \times 2$. b) One-stage equivalent for $M = 8 \times 2$.

TABLE II
DECIMATOR IMPLEMENTATIONS FOR $M = 16$.

J	N_1	N_2	M_1	M_2	R_T	N_T
1	463	-	16	-	3.6328125×10^9	463
2	44	61	8	2	1.1640625×10^9	105
2	13	124	4	4	1.3750000×10^9	137

TABLE III
OVERSAMPLED DECIMATOR IMPLEMENTATIONS FOR $M = 8$.

J	N_1	N_2	N_3	M_1	M_2	M_3	R_T	N_T
1	92	-	-	16	-	-	1.437500×10^9	92
2	32	23	-	4	2	-	1.359375×10^9	55
3	12	15	24	2	2	2	1.593750×10^9	51

is decimated by 8 by the filter $H_1(z)$, and in the second stage by 2 for the filter $H_2(z)$. Fig. 4(b) presents the equivalent one-stage arrangement of the two-stage decimator, obtained after applying the third Noble identity [25], [26]. The function of the $H_1(z)$ and $H_2(z)$ filters is to enforce the general design requirements specified for the decimator. The $H_2(z)$ filter, operating at the 15.625 MHz sampling rate, is designed for the 3.125 MHz passband cut-off frequency and the 11.719 MHz edge frequency. The role of $H_1(z)$ is to provide a passband in the $0 - 3.125$ MHz range and to provide the 66 dB attenuation in the rest of the spectrum. With this approach, the overall filtering task is shared between two lower-order filters. The advantages of the two-stage decimator are apparent when comparing the filter characteristics with those required for the one-stage decimator. Table II shows the parameters found for the implementations of the decimator by $M = 16$. The memory requirements, N_T, for each implementation have also been included.

To complete the analysis, the filterbank design oversampled by a factor $I = 2$ and $K = MI = 16$ channels was also analysed. Fig. 5 and Table III show the results.

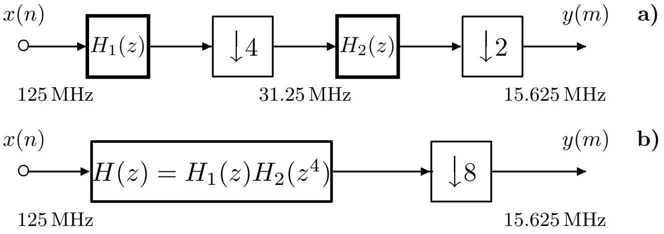


Fig. 5. Two-stage oversampled decimator. a) Implementation for $M = 4 \times 2$. b) One-stage equivalent for $M = 4 \times 2$.

VI. ANALYSIS FOR $K \geq 10.000$ CHANNELS

The challenge is to efficiently read an array of 1000 or more superconducting detectors on an FPGA that has limited hardware resources. In this section we analyse the different possible implementations to be able to read these large arrays of detectors, starting from 512 MHz up to 2 MHz as these are typical values in nowadays hardware. That is, designs with decimation factors of $M = 256$, critically sampled and $M = 128$, oversampled by a factor $I = 2$. Table IV and Table V show the results of the available M_i factors for the critically sampled and oversampled cases, respectively.

TABLE IV
AVAILABLE M_i FACTORS FOR $M = 256$ AND $J = 1, 2, 3, 4$.

J	M_i factors
1	256
2	(64, 4), (128, 2), (16, 16), (32, 8)
3	(16, 8, 2), (16, 4, 4), (32, 4, 2), (8, 8, 4), (64, 2, 2)
4	(8, 4, 4, 2), (4, 4, 4, 4), (16, 4, 2, 2), (8, 8, 2, 2), (32, 2, 2, 2)

TABLE V
AVAILABLE M_i FACTORS FOR $M = 128$ AND $J = 1, 2, 3, 4$.

J	M_i factors
1	128
2	(16, 8), (32, 4), (64, 2)
3	(8, 8, 2), (32, 2, 2), (16, 4, 2), (8, 4, 4)
4	(16, 2, 2, 2), (8, 4, 2, 2), (4, 4, 4, 2)

To conclude the analysis of efficiency in multi-stage systems, Fig. 6 shows a comparison between the critically sampled and oversampled designs that have been analyzed up to now. It can be seen from the figure that although the critically sampled design has a higher computational load when implemented in a single stage, it quickly outperforms the oversampled design when implemented in multiple stages. However, if the design is to be implemented in a single stage, it is better to choose oversampling. It is also true that for the readout of micro-resonators for spectral studies, it is best to use an oversampled system, since in this way there is no loss of spectral zones. Taking this last into account, it can be seen in the figure that there is an important change when going from one stage to two stages, but then not only that there is not much difference in computational efficiency, but also that it decreases (or what is the same, the computational load increases).

In terms of memory element usage, the oversampled design is more efficient in designs up to two stages. For more stage designs, the difference in memory resource usage between the two implementations is negligible.

VII. CONCLUSIONS

The problem of optimizing filterbanks was addressed in order to understand the factors that define their efficiency. We presented the analysis of structures in stages, as possible option for the development of the readout system.

Thanks to the algorithm developed to find the optimal number of stages, it was possible to find the most efficient implementation. Although the critically sampled design has a higher computational burden when implemented in a single stage, it quickly outperforms the oversampled design when implemented in multiple stages.

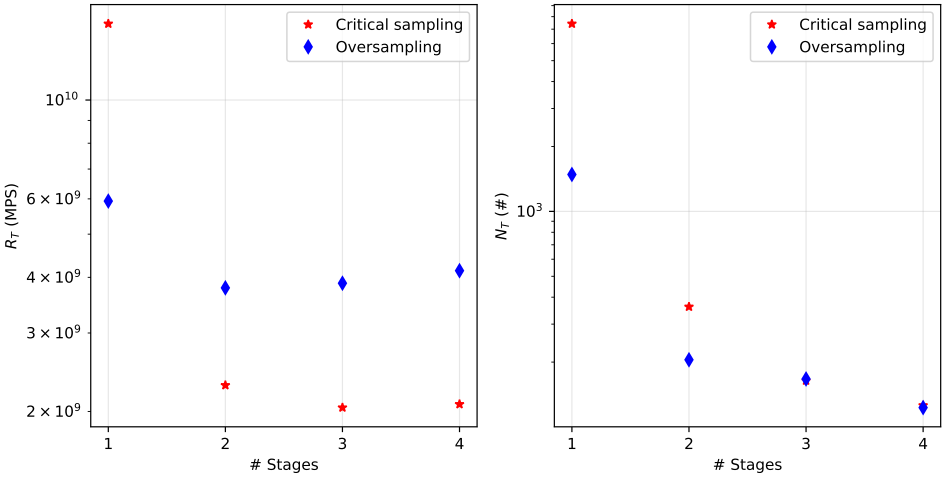


Fig. 6. Comparison of resource use and computational efficiency between critically sampled and oversampled designs (I = 2). Designs with $F_{s,0} = 512\,\text{MHz}$ and $K = 256$ are compared.

If the choice is to implement the filterbank in a single stage, it is convenient to use the oversampled system.

In terms of memory element usage, the oversampled alternative is more efficient in designs with up to two stages. For multi-stage designs, the difference between both implementations is negligible.

References

[1] F. Harris, C. Dick, and M. Rice, "Digital receivers and transmitters using polyphase filter banks for wireless communications," *IEEE Trans. Microw. Theory Tech.*, vol. 51, no. 4, pp. 1395–1412, apr 2003.

[2] F. J. Harris, *Multirate Signal Processing for Communication Systems*. Prentice Hall, may 2004, vol. 1.

[3] J. Van Rantwijk, M. Grim, D. Van Loon, S. Yates, A. Baryshev, and J. Baselmans, "Multiplexed Readout for 1000-Pixel Arrays of Microwave Kinetic Inductance Detectors," *IEEE Trans. Microw. Theory Tech.*, vol. 64, no. 6, pp. 1876–1883, 2016.

[4] L. H. Arnaldi and H. D. Dellavale, "Oversampled filter bank channelizer for cryogenic detectors," *Review of Scientific Instruments*, vol. 92, no. 2, p. 023304, feb 2021.

[5] J. Tuthill, G. Hampson, J. Bunton, A. Brown, S. Neuhold, T. Bateman, L. De Souza, and J. Joseph, "Development of multi-stage filter banks for ASKAP," *Proc. 2012 Int. Conf. Electromagn. Adv. Appl. ICEAA'12*, pp. 1067–1070, 2012.

[6] P. K. Day, H. G. LeDuc, B. A. Mazin, A. Vayonakis, and J. Zmuidzinas, "A broadband superconducting detector suitable for use in large arrays." *Nature*, vol. 425, no. 6960, pp. 817–21, oct 2003.

[7] B. A. Mazin, B. Bumble, P. K. Day, M. E. Eckart, S. Golwala, J. Zmuidzinas, and F. A. Harrison, "Position sensitive x-ray spectrophotometer using microwave kinetic inductance detectors," *Appl. Phys. Lett.*, vol. 89, no. 22, p. 222507, 2006.

[8] L. H. Arnaldi, "Técnicas Avanzadas de procesamiento digital con aplicaciones en microresonadores superconductores multipíxeles," 2021. [Online]. Available: https://ricabib.cab.cnea.gov.ar/985/1/Arnaldi.pdf

[9] ——, "Implementation of a Polyphase Filter Bank Channelizer on a Zynq FPGA," in *2020 Argentine Conf. Electron.*, no. 978. IEEE, feb 2020, pp. 57–62.

[10] O. Herrmann, L. R. Rabiner, and D. S. K. Chan, "Practical Design Rules for Optimum Finite Impulse Response Low-Pass Digital Filters," *Bell Syst. Tech. J.*, vol. 52, no. 6, pp. 769–799, jul 1973.

[11] R. Crochiere and L. Rabiner, "Optimum FIR digital filter implementations for decimation, interpolation, and narrow-band filtering," *IEEE Trans. Acoust.*, vol. 23, no. 5, pp. 444–456, oct 1975.

[12] STEMLab RedPitaya, "RedPitaya Open Source Instrument." [Online]. Available: http://www.redpitaya.com/

[13] M. G. Bellanger, G. Bonnerot, and M. Coudreuse, "Digital Filtering by Polyphase Network: Application to Sample-Rate Alteration and Filter Banks," *IEEE Trans. Acoust.*, vol. 24, no. 2, pp. 109–114, 1976.

[14] P. P. Vaidyanathan, "Multirate Digital Filters, Filter Banks, Polyphase Networks, and Applications: A Tutorial," *Proc. IEEE*, vol. 78, no. 1, pp. 56–93, 1990.

[15] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*. Prentice-Hall, may 1983, vol. 1.

[16] M. Vetterli, "A Theory of Multirate Filter Banks," *IEEE Trans. Acoust.*, vol. 35, no. 3, pp. 356–372, 1987.

[17] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing*. Pearson, abr 2006, vol. 1.

[18] L. Rabiner, "Approximate design relationships for low-pass FIR digital filters," *IEEE Trans. Audio Electroacoust.*, vol. 21, no. 5, pp. 456–460, oct 1973.

[19] M. Coffey, "Optimizing multistage decimation and interpolation processing," *IEEE Signal Process. Lett.*, vol. 10, no. 4, pp. 107–110, apr 2003.

[20] M. W. Coffey, "Optimizing Multistage Decimation and Interpolation Processing—Part II," *IEEE Signal Process. Lett.*, vol. 14, no. 1, pp. 24–26, jan 2007.

[21] Der-Feng Huang, "The direct integer factorization approach to the Crochiere and Rabiner multistage FIR designs for multirate systems," in *3rd Int. Symp. Image Signal Process. Anal. 2003. ISPA 2003. Proc.*, vol. 2. IEEE, 2004, pp. 1060–1065.

[22] D.-F. Huang and S.-R. Hung, "The Optimum Design of Multistage Multirate FIR Filter for Audio Signal Sampling Rate Conversion via a Genetic Algorithm Approach," in *2009 2nd Int. Congr. Image Signal Process.*, vol. 2. IEEE, oct 2009, pp. 1–5.

[23] X. Zhu, Y. Wang, W. Hu, and J. D. Reiss, "Practical considerations on optimising multistage decimation and interpolation processes," in *2016 IEEE Int. Conf. Digit. Signal Process.*, vol. 0, no. 4. IEEE, oct 2016, pp. 370–374.

[24] R. E. Crochiere and L. R. Rabiner, "Interpolation and Decimation of Digital Signals-A turorial review," *IEEE Trans. Geosci. Remote Sens.*, vol. 69, no. 3, pp. 300–331, 1981.

[25] ——, *Multirate Digital Signal Processing*. Pearson, mar 1983, vol. 1.

[26] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*. Prentice Hall, oct 1993, vol. 1.

Real-time noise reduction through independent channel averaging for real-time biomedical signal acquisition

1st Federico N. Guerrero
LEICI (UNLP-CONICET-CIC),
La Plata, Argentina.
federico.guerrero@ing.unlp.edu.ar

2nd Matías Oliva
LEICI (UNLP-CONICET-CIC),
La Plata, Argentina.
matias.oliva@ing.unlp.edu.ar

3rd Enrique M. Spinelli
LEICI (UNLP-CONICET-CIC)
La Plata, Argentina.
spinelli@ing.unlp.edu.ar

***Abstract*—In this work, a strategy to obtain a lower noise floor from commercial multichannel Sigma-Delta analog-to-digital converters (ADCs) is presented. Specifically, data from ADS131E08, an 8-channel simultaneous sampling 24 bit converter, is captured, processed, and transmitted in real time using a MAX 10 FPGA included in a measurement system with medical grade isolation, thus able to acquire biomedical signals. Noise measurements show that the system is able to reduce the equivalent input voltage noise of the ADC by a factor of 2.8, extending the measurement dynamic range by 9 dB. In this way, the system improves the otherwise minimum available noise floor with no additional analog stages and allows using higher data rates while maintaining signal quality. Experimental electrocardiogram and electromyogram recordings were taken using non-invasive dry electrodes, validating the operation of the system as a biopotential acquisition platform. Under these experimental conditions, a noise reduction factor of 2.1 times for the noise floor of the measured biopotential signals was verified.**

***Index Terms*—noise, biopotential, average, fpga, sigma-delta converter, dynamic range**

I. Introduction

Sigma-Delta ($\Sigma\Delta$) analog-to-digital converters (ADCs) provide a unique solution for biomedical signal acquisition since their output data stream can achieve a very high dynamic range (DR) and very low noise. When biomedical signals are measured from the body, the transducing electrodes introduce a DC offset with a $V_{off,max} = \pm 300\,\mathrm{mV}$ range. The $V_{off,max}$ value is taken from electrocardiography (ECG) standards for a differential channel [1] and although it is pessimistic, it is often the case that the offset is within a $10\,\mathrm{mV}$ to $100\,\mathrm{mV}$ range. On the other hand, biopotential signals should be measured with a noise floor on the order of $1\,\mathrm{\mu V_{rms}}$ or less [2], [3] which can be translated to a signal amplitude of $6\,\mathrm{\mu V_{pp}}$ considering $\pm 3\sigma$ thus taking the dynamic range to $20\log_{10}(600\,\mathrm{mV}/6\,\mathrm{\mu V}) = 100\,\mathrm{dB}$.

If high-pass filtering is applied the dynamic range is reduced by blocking the DC offset decreasing the necessary DR to the approximately 70 dB needed by the signal itself. In these cases, the usual strategy is to apply filtering in combination with a relatively high amplification factor to allow acquisition with a $12\,\mathrm{bit}$ to $14\,\mathrm{bit}$ ADC. However, there are a number of disadvantages to this methodology. The first disadvantage is simply the necessity of an analog stage capable of providing a relatively high gain in the order of 100-1000 times while rejecting DC electrode offset components [4]. The second is related to a form of interference affecting biomedical measurements called *artifacts* which are produced when the electrodes or the skin are mechanically perturbed (e.g. by pulling the cables or by the displacement of the muscles themselves). Artifacts produce wide fluctuations of the baseline signal which can saturate filters and amplification stages, that in turn may take a long time to return to an operational range. Moreover, front-end solutions with additional amplification may require further complex balancing circuits [5] compared with unity-gain solutions [6].

Thus, high dynamic range, low noise $\Sigma\Delta$ converters have been identified as advantageous for biomedical signal measurements [7] and this strategy is incorporated in state-of-the-art integrated acquisition systems [8], [9]. The semiconductor industry has included these ADCs in commercial application-specific standard products (ASSPs) such as the ADS129x and ADS13x lines from Texas Instruments and AD7779 from Analog Devices, as some examples. These ASSPs are useful in programmable-logic-based systems as biomedical signal front-ends [10], [11].

Sigma-Delta converters are among the highest DR ADCs commercially available and achieve a very low referred-to-the-input (RTI) noise voltage. The RTI noise is in fact sufficiently low to enable biomedical signals to be acquired with no amplification, preserving the full DR of the device. However, augmenting the output data rate (ODR) results in a degradation of the noise properties of these devices because of their fundamental trade-off between resolution and bandwidth [12]. This trade-off can be seen in Table I, where values obtained from Table 1 and Equation 1 of ADS131E08's datasheet [13] are displayed. The motivation of the presented work is to further extend the capabilities of commercial $\Sigma\Delta$ ADC devices making (i) previously unavailable noise-floor levels feasible without additional analog stages and (ii) lower noise levels at

This work was funded by projects CONICET PIP-0323, UNLP PID/I254 and PPID/I016, and PICT-2018/3747.

TABLE I: ADS131E08 specifications parametrized by gain and output data rate (from its datasheet [13])

Gain - ODR*	1 - 1 kHz	12 - 1 kHz	1 - 16 kHz	12 - 16 kHz
DR [dB]	117.7	108.0	102.8	94.2
ENOB † [bits]	19.6	18.0	17.07	15.65
RTI N. ‡ [μV$_{\mathrm{rms}}$]	2.13	0.54	12.33	2.75

*Output Data Rate.
† Effective number of bits.
‡ Referred-to-the-input noise voltage.

higher data rates available.

II. Method

A. Signal-to-noise ratio improvement through averaging

One strategy to improve the signal-to-noise ratio (SNR) of an ADC is to use oversampling by which temporal averaging is used to reduce noise. However, this technique implies a reduction of the ODR. If several independent ADC channels are available, the SNR can be improved by simultaneously acquiring the same signal with all channels and then averaging across all digitized samples [14], [15] as shown in Fig. 1.

Considering a signal $v_{in}(t)$ which is sampled obtaining

$$x(k) = v_{in}(kT) + n_{ADC}(k) = s(k) + n_{ADC}(k) \tag{1}$$

where n_{ADC} is the noise introduced by the analog-to-digital conversion process and $s(k)$ is the sampled signal of interest, the output of the N-channel averaging stage is equal to

$$y(k) = \frac{1}{N}\sum_{i=1}^{N} x_i(k). \tag{2}$$

After applying (2) to the N input signals given by (1) then

$$\begin{aligned} y(k) &= \frac{1}{N}\sum_{i=1}^{N}\left(s_i(k) + n_{ADC,i}(k)\right) \\ &= s(k) + \frac{1}{N}\sum_{i=1}^{N} n_{ADC,i}(k) = s(k) + n_T(k) \end{aligned} \tag{3}$$

The output $y(k)$ contains the unmodified component of the signal of interest $s(k)$ and the averaged noise from all channels $n_T(k)$. While for ideal ADCs quantization noise is linked to the weight of the least significant bit (LSB), the noise contribution of $\Sigma\Delta$ converters is measured by an equivalent number of bits (ENOB) which corresponds to the root mean square (rms) noise level and is dependent on the output data-rate and the digital filter implementation.

Given the assumption that the noise is an independent identically distributed (iid) random process with Gaussian distribution, the rms value can be obtained by calculating its standard deviation σ_{ADC}, which can then be further estimated by the standard deviation over sampled time k under the assumption of ergodicity.

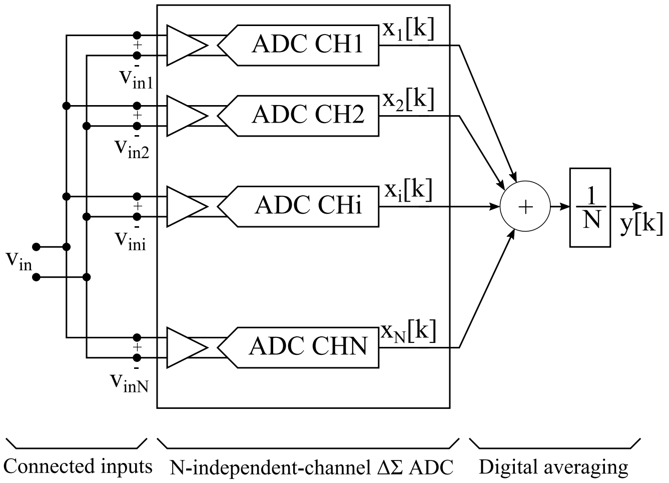


Fig. 1: Block diagram of the simultaneous sampling averaging scheme.

The total noise component present in the output signal will have a variance given at each instant k by

$$\begin{aligned} E\{n_T(k)^2\} = \sigma_T^2 &= E\left\{\left(\frac{1}{N}\sum_{i=1}^{N} n_{ADC,i}\right)^2\right\} \\ &= \frac{1}{N^2}\sum_{i=1}^{N}\sigma_{ADC}^2 + \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N,i\neq j}\rho_{i,j}\sigma_i\sigma_j \end{aligned} \tag{4}$$

Where $\rho_{i,j}$ is the correlation between the noise of pairs of channels and $E\{n_{ADC,i}\}$ has been considered 0. The best case is given if all channels are uncorrelated and hence $\rho_{i,j} = 0\,\forall\, i \neq j$ therefore

$$\sigma_T^2 = \sigma_{ADC}^2/N \;\therefore\; \sigma_T = \sigma_{ADC}/\sqrt{N} \tag{5}$$

yielding a SNR improvement equal to $\sqrt{N}$ [14], [16]. The worst case is given when the correlation between all noise signals is 1 in which case $\sigma_T = \sigma_{ADC}$ and the SNR is exactly the same as if no averaging had been performed.

B. System implementation

An acquisition system was built in order to capture data from an ADS131E08 ADC and perform the average of its channels per (2). The system was implemented as a full biopotential acquisition platform including medical grade isolation in order to perform biopotential measurements. Its main blocks and their interconnections can be seen in Fig. 2.

The ADS131E08 has 8 independent $\Sigma\Delta$ ADCs, each with a programmable gain amplifier (PGA). The 8 channels perform simultaneous sampling of each of their differential inputs. The output data rate is configurable from 1 kHz to 16 kHz. Higher 32 kHz and 64 kHz data rates are available but the dynamic range is reduced to less than 16 bits, therefore they were not used.

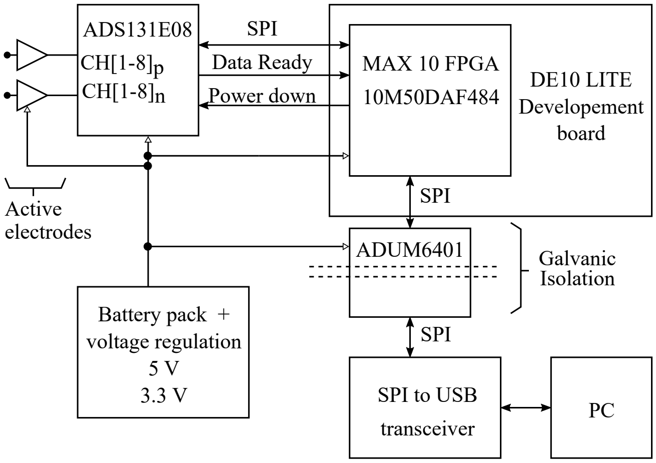


Fig. 2: Block diagram of the biopotential acquisition system implementation.

The core element controlling the system is a MAX 10 10M50DAF484C7G FPGA from Intel (previously Altera) used in a DE10 LITE development board from Terasic. The rationale for selecting an FPGA instead of a microcontroller platform was that for the latter the volume and rate of data can represent a demanding task. In contrast, an FPGA would be scarcely demanded (as will be shown later in this work) and has the potential to further develop both the technique by including more complex processing, and the acquisition platform by including user and data interface capabilities.

The system was configured with a set of finite state machines (FSMs) controlling a communications interface to program ADS131 registers and receive its data output, summing logic to perform the average, and a second communications interface to send the data in real time to a personal computer (PC). The ADS131 interface consists of a master serial peripheral interface (SPI) bus plus control lines required by the converter. The interface with the PC is achieved through a second master SPI bus sending the processed output data in a frame for serialized transmission. The behavior and function of the configured FSMs are shown in Fig. 3. A $50\,\mathrm{MHz}$ clock included in the DE10 Lite board was used as input to a phase-locked loop (PLL) to obtain a $100\,\mathrm{MHz}$ master clock signal. The SPI master module was sourced from OpenCores SPI MASTER/SLAVE project [17].

C. Experimental setup

In order to ascertain the viable noise reduction that can be obtained with the proposed technique, all channels of the ADS131E08 ADC were short-circuited to a voltage reference using the internal multiplexer of the device. The resulting RTI noise was evaluated by taking $32\,000$ samples at $1\,\mathrm{kHz}$ and $16\,\mathrm{kHz}$ for all available gains (1, 2, 4, 8 and 12), and at gains 1 and 12 for all available output data rates (1, 2, 4, 8 and 16 kHz). Results were compared with data extracted from Table 1 and Equation 1 of ADS131E08's datasheet [13].

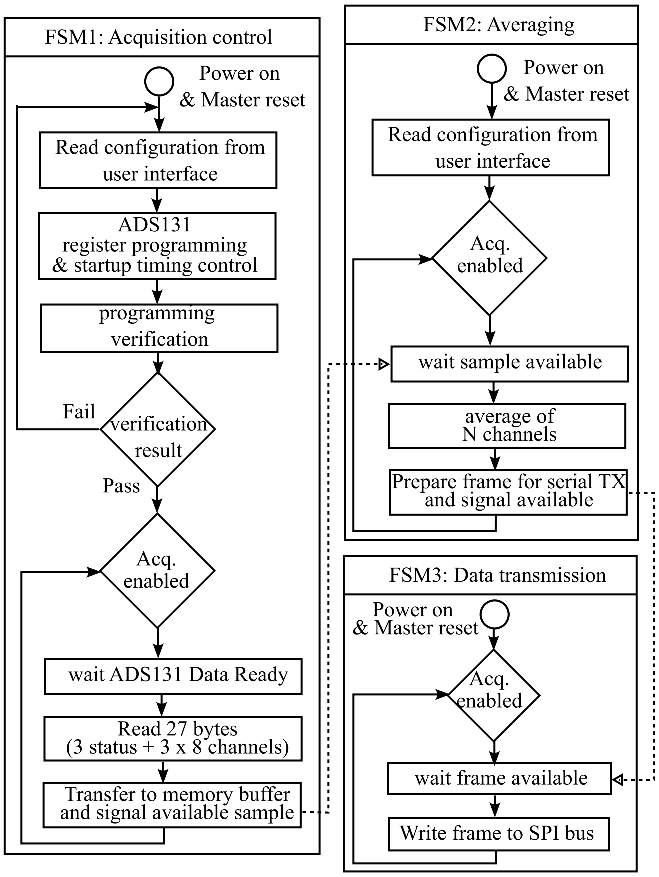


Fig. 3: Programmable-logic system configuration.

Biopotential measurements were carried out to validate the usefulness of the acquisition system and noise-reduction impact. A previously reported driven-right-leg (DRL) independent electrode [18] was connected to the isolated power provided by the board with a reference of $1.2\,\mathrm{V}$. Measurements were performed by attaching two active electrodes composed of operational amplifiers (OAs) in unity-gain buffer configuration to the positive and negative inputs of the ADS131 ADC respectively. Therefore, one differential channel can be measured with the active electrodes shown in Fig. 2 and routed to v_{in} signal shown in more detail in Fig. 1, thereby connecting it to the 8 inputs simultaneously. The buffers in the active electrodes serve to mitigate electromagnetic interference (EMI) due to capacitance couplings to the wires as is their usual function [19] and in this case, are paramount to avoid input impedance degradation due to the connection to multiple input stages in parallel.

III. Results

A. System implementation

The implemented acquisition system is shown in Fig. 4 with the blocks from Fig. 2 marked with text commentary. The

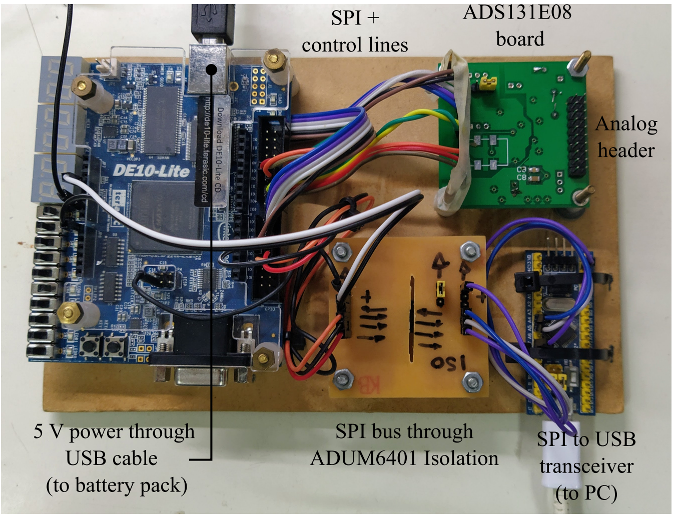


Fig. 4: Photograph of the acquisition system.

Resource utilization	
Total logic elements	1,845 / 49,760 (4 %)
Total registers	1446
Total memory bits	640 / 1,677,312 (<1 %)
Total PLLs	1 / 4 (25 %)
Timing Analysis	
Maximum Frequency	125.2 MHz

TABLE II: Implementation details for 10M50DAF484C7G device.

summary of the total utilized resources of the programmable-logic device is shown in table II. The timing results were obtained with the Timing Analyzer tool from Intel's Quartus Prime software, for a 1200 mV 85C model.

B. Noise measurements

Noise measurement results are sown in Fig. 5 and Fig. 6.

In Fig. 5 two sets of values of the measured RTI noise are shown joined by a full line: they correspond to the noise of one channel at 1 kHz and 16 kHz ODRs for its 5 gain configurations. A higher gain results in lower RTI noise since the contribution of stages after the programmable gain amplifier has less weight. The effective noise reported in the component's datasheet is shown in small dots and it coincides up to within a 5 % with all measurements except the 16 kHz, 12x gain which consistently deviated with a 13 % lesser noise. In the same figure, the results of the averaged measurements are shown in dashed line, verifying that a noise reduction was indeed obtained. At a 1 kHz data rate, noise was reduced by a factor of 2.9 ± 0.1, and at 16 kHz by 2.97 ± 0.08.

Noise measurements from the lower and higher gain configurations (1 and 12) are further shown in Fig. 6, again joined by a full line for the RTI noise of the single-channel case, with markers at the 5 data rate configurations and small point markers showing the component's datasheet values. An average noise reduction of 2.80 ± 0.05 was obtained for gain 1, and 3.04 ± 0.02 for gain 12.

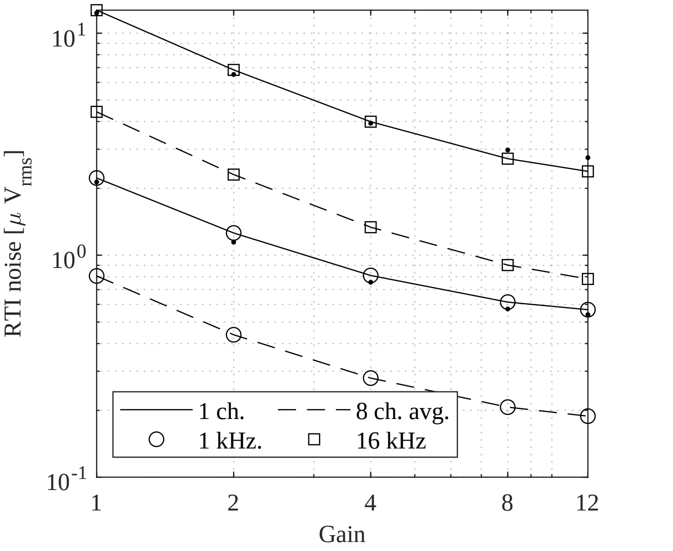


Fig. 5: Measured RTI noise for 1 kHz and 16 kHz data rates at different gain configurations. The full line follows measurements of a single channel and the dashed line the 8 channels averaged.

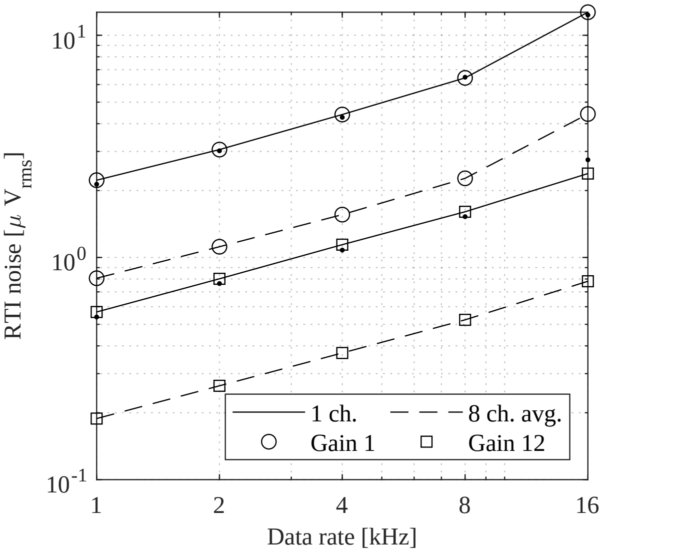


Fig. 6: Measured RTI noise for gain of 1 and 12 at different data-rate configurations. The full line follows measurements of a single channel and the dashed line the 8 channels averaged.

The results from Fig. 5 and Fig. 6 show that for ADS131E08 it is possible to reduce noise by channel averaging. The reduction factor is slightly higher than the expected for the 8-channel average ($\sqrt{8} = 2.82$) and could be explained by a deviation from the simplified statistical assumptions made, and the necessary use of an estimation of the standard deviation. However, it is an indicator that there are no inter-channel correlations preventing noise reduction. The lowest measured reduction coincides with the theoretical value, and it effectively extends the dynamic range by 9 dB.

The absolute noise levels obtained by averaging show that for biomedical signal measurement applications, no additional amplification circuit would be needed at a frequency as high as $16\,\mathrm{kHz}$ to attain $1\,\mu\mathrm{V}_{\mathrm{rms}}$, since all values for a gain of 12 are below this limit. Moreover, a gain of 8 is sufficient to achieve the target noise level at the $16\,\mathrm{kHz}$ rate. This is significant since this data rate allows acquiring fast spike potentials with a $4\,\mathrm{kHz}$ bandwidth present in invasive measurements [20]. The $9\,\mathrm{dB}$ DR extension for the fastest data rate takes the DR to $103\,\mathrm{dB}$ again above the desired values. On the other hand, the low noise levels achieved on the lowest frequency settings, for example, $0.19\,\mu\mathrm{V}_{\mathrm{rms}}$ for a $1\,\mathrm{kHz}$ ODR at gain 12 while preserving a $117\,\mathrm{dB}$ DR, can be useful for electroencephalography (EEG) measurements which present challenging low-noise requirements [2] and in some cases benefit from DC-coupled acquisition [21].

C. Biopotential measurements

In order to validate the operation of the system for real-time biopotential acquisition, a set of in-vivo measurements were performed.

First, an electrocardiogram (ECG) recording was taken using a differential channel with active electrodes implemented with OPA378 operational amplifiers (OAs) in buffer configuration (gain of 1). Two standard adhesive wet Ag/AgCl electrodes were placed frontally below the thorax and the DRL on the waist. The system was configured with an output data rate of $1\,\mathrm{kHz}$ and gain of 1. Measurements were taken 10 minutes after attaching the electrodes. The channel averaging was first turned off producing the upper line from Fig. 7, and then it was activated to perform the 8-channel average. The result is shown in the lower trace of Fig. 7. In this case, the noise floor was imposed by the measurement electrodes, and the signals show a match validating the averaged configuration acquisition capability.

Next, electromyography (EMG) measurements were conducted by placing two dry electrodes on the forearm, affixed with an elastic fabric band, and performing finger contractions. The system was configured with a $16\,\mathrm{kHz}$ output data rate and gain of 1. Fig. 8 shows the measurement results. A very slight contraction was performed between seconds 0 and 2, and a stronger contraction between seconds 5 and 8. An effort to relax the muscles was instructed between these two contractions, and within these segments the noise reduction is observable despite the presence of the base noise floor of the dry electrodes. In order to observe the properties of the system, different passband filters were applied in Fig. 8a and 8b. In 8a a low-pass frequency of $4\,\mathrm{kHz}$ was used and the noise reduction obtained with averaging is seen by inspection, with a reduction factor of 2.1. Under these conditions, the activity resulting from weaker contractions is not observable. In Fig. 8b, standard band-pass filtering for superficial EMG was used [3] and the signal quality improves, with a reduction gained by averaging of 1.12. The bandwidth used in Fig. 8a is excessive for superficial EMG but would be useful in invasive EMG or electroneurography (ENG) measurements where the full bandwidth of biopotential spikes up to $10\,\mathrm{kHz}$ can be acquired [20].

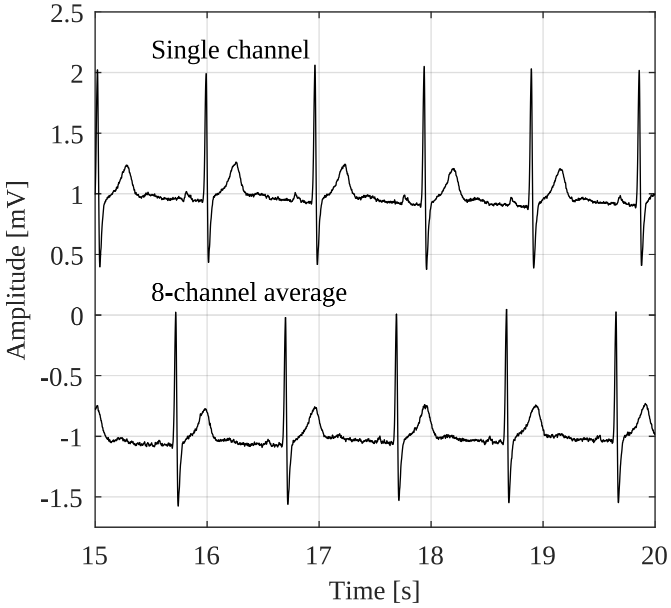


Fig. 7: Recording from ECG measurements.

IV. Conclusions

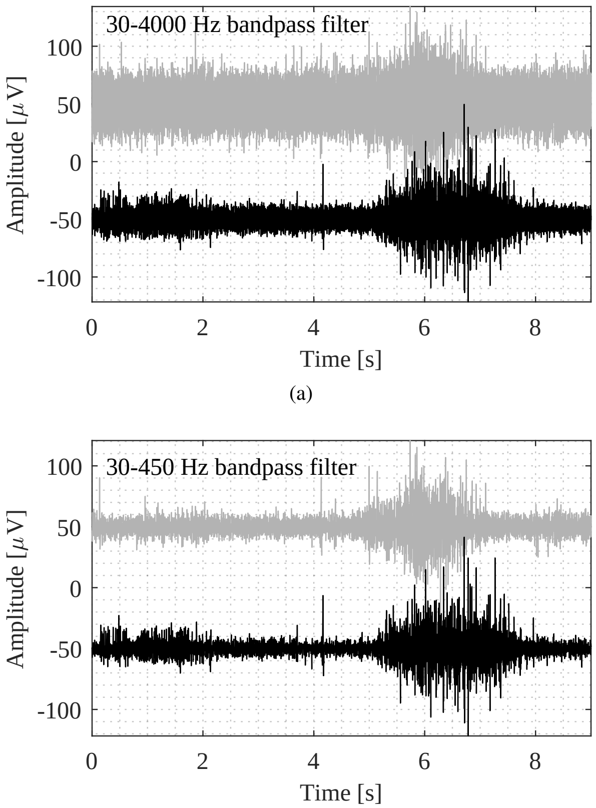
Noise reduction through the averaging of $N = 8$ channels of a simultaneous sampling high-resolution Delta-Sigma analog-to-digital converter was shown to be feasible and in the order of the theoretical reduction for independent identically distributed noise sources, $\sqrt{N}$. The capabilities of the implemented system for real-time biomedical signal acquisition and processing were demonstrated by a set of ECG and EMG in-vivo recordings.

Considering the recommended noise floor for biomedical signal acquisition, the application of this method for ADS131E08 $\Sigma\Delta$ ADC allows measuring with no additional analog amplification besides the integrated programmable-gain-amplifier for data rates of $4\,\mathrm{kHz}$ to $16\,\mathrm{kHz}$, using a gain setting of 12. Further, acquisition with a configuration of $1\,\mathrm{kHz}$ data rate and gain of 1 is possible thus preserving the full available dynamic range. In addition, the lowest noise floor (for $1\,\mathrm{kHz}$ data rate and gain of 12) was reduced to $0.18\,\mu\mathrm{V}_{\mathrm{rms}}$.

The presented method allowed reducing the equivalent input voltage noise of the ADC by a factor of 2.8, extending the measurement dynamic range by 9 dB. This technique can thus be useful to reduce the noise floor in measurements using $\Sigma\Delta$ converters with no additional analog stages and to achieve higher data rates without sacrificing signal quality.

References

[1] AAMI, "Medical Electrical Equipment - Part 2-25: Particular Requirements For The Basic Safety And Essential Performance Of Electrocardiographs," Association for the Advancement of Medical Instrumentation, Standard ANSI/AAMI/IEC 60601-2-25:2011 (R2016), 2016.



(a)

(b)

Fig. 8: Superficial EMG measurement recordings. The gray line marks a single-channel measurement and the black line marks a measurement performed using averaging. (a) and (b) have different pass-band filters applied as described in the text.

[2] J. J. Halford, D. Sabau, F. W. Drislane, T. N. Tsuchida, and S. R. Sinha, "American Clinical Neurophysiology Society Guideline 4: Recording Clinical EEG on Digital Media," *The Neurodiagnostic Journal*, vol. 56, no. 4, pp. 261–265, Oct. 2016.

[3] R. Merletti and G. Cerone, "Tutorial. Surface EMG detection, conditioning and pre-processing: Best practices," *Journal of Electromyography and Kinesiology*, vol. 54, p. 102440, Oct. 2020.

[4] F. N. Guerrero and E. M. Spinelli, "Chapter 4: Biopotential acquisition systems," in *Medicine-Based Informatics and Engineering*, F. Simini and P. Bertemes-Filho, Eds. Cham: Springer International Publishing, 2022, pp. 51–79.

[5] T. Degen and H. Jäckel, "Enhancing interference rejection of preamplified electrodes by automated gain adaption." *IEEE transactions on bio-medical engineering*, vol. 51, no. 11, pp. 2031–9, Nov. 2004.

[6] F. N. Guerrero and E. M. Spinelli, "A two-wired ultra-high input impedance active electrode," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, no. 2, pp. 437–445, 2018.

[7] D. Berry, F. Duignan, and R. Hayes, "An Investigation of the use of a High Resolution ADC as a Digital Biopotential Amplifier," *4th European Conference of the International Federation for Medical and Biological Engineering*, pp. 0–6, 2009.

[8] X. Yang, J. Xu, M. Ballini, H. Chun, M. Zhao, X. Wu, C. Van Hoof, C. M. Lopez, and N. Van Helleputte, "A 108 dB DR $\Delta - \Sigma$-ΣM Front-End With 720 mV pp Input Range and$>$ ±300 mV Offset Removal for Multi-Parameter Biopotential Recording," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 15, no. 2, pp. 199–209, 2021.

[9] Y. Jung, S. Kweon, H. Jeon, I. Choi, J. Koo, M. K. Kim, H. J. Lee, S. Ha, and M. Je, "A wide-dynamic-range neural-recording ic with automatic-gain-controlled afe and ct dynamic-zoom $\delta\sigma$ adc for saturation-free closed-loop neural interfaces," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 10, pp. 3071–3082, 2022.

[10] J. Chen, X. Li, X. Mi, and S. Pan, "A high precision eeg acquisition system based on the compactpci platform," in *2014 7th International Conference on Biomedical Engineering and Informatics*. IEEE, 2014, pp. 511–516.

[11] D. Liu, Q. Wang, Y. Zhang, X. Liu, J. Lu, and J. Sun, "Fpga-based real-time compressed sensing of multichannel eeg signals for wireless body area networks," *Biomedical Signal Processing and Control*, vol. 49, pp. 221–230, 2019.

[12] P. M. Aziz, H. V. Sorensen, and J. Van der Spiegel, "An overview of sigma-delta converters: How a 1-bit ADC achieves more than 16-bit resolution," *IEEE Signal Processing Magazine*, vol. 13, no. 1, pp. 61–84, 1996.

[13] *ADS131E0x 4-, 6-, and 8-Channel, 24-Bit, Simultaneously-Sampling, Delta-Sigma ADC*, Texas Instruments, 2017. [Online]. Available: https://www.ti.com/document-viewer/ads131e08/datasheet

[14] O. Rompelman and H. Ros, "Coherent averaging technique: A tutorial review Part 1: Noise reduction and the equivalent filter," *Journal of Biomedical Engineering*, vol. 8, no. 1, pp. 24–29, Jan. 1986.

[15] E. B. Loewenstein, "Reducing the Effects of Noise in a Data Acquisition System by Averaging," National Instruments, Application Note AN152, 2000.

[16] D. S. Lemons and P. Langevin, *An introduction to stochastic processes in physics: containing "On the theory of Brownian motion" by Paul Langevin, translated by Anthony Gythiel*. Baltimore: Johns Hopkins University Press, 2002, oCLC: ocm47716485.

[17] Jonny Doin, "SPI MASTER / SLAVE INTERFACE," Aug. 2011. [Online]. Available: https://opencores.org/projects/spi_master_slave

[18] F. N. Guerrero and E. Spinelli, "High gain driven right leg circuit for dry electrode systems," *Medical Engineering and Physics*, vol. 39, pp. 117–122, Jan. 2017, publisher: Elsevier Ltd.

[19] S. Nishimura, Y. Tomita, and T. Horiuchi, "Clinical application of an active electrode using an operational amplifier," *IEEE Trans. Biomed. Eng.*, vol. 39, no. 10, pp. 1096–1099, 1992.

[20] D. B. Sanders, K. Arimura, L. Cui, M. Ertaş, M. E. Farrugia, J. Gilchrist, J. A. Kouyoumdjian, L. Padua, M. Pitt, and E. Stålberg, "Guidelines for single fiber EMG," *Clinical Neurophysiology*, vol. 130, no. 8, pp. 1417–1439, Aug. 2019.

[21] P. Tallgren, S. Vanhatalo, K. Kaila, and J. Voipio, "Evaluation of commercially available electrodes and gels for recording of slow EEG potentials." *Clinical neurophysiology : official journal of the International Federation of Clinical Neurophysiology*, vol. 116, no. 4, pp. 799–806, Apr. 2005.

Diseño en VHDL del algoritmo SOGI PLL SRF usando síntesis de alto nivel (HLS)

Alejandro Núñez Manquez
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
janyo12@gmail.com

Matín Murdocca
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
mmurdocc@gmail.com

Victor Yelpo
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
victoryelpo@gmail.com

Ivana Trento
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
trentoivana@gmail.com

***Abstract*—Los algoritmos de detección de fase son muy utilizado para la sincronización de inversores con la red eléctrica para la inyección de potencia a la misma. Estos algoritmos, que generalmente están basados en PLL (Phase-Locked Loop) o lazo de seguimiento de fase, permiten generar una señal cuya fase está acoplada con la fase de la señal de entrada. A estos algoritmos se les suma en su diseño el algoritmo SOGI (Second-order Generalized Integrators) y transformadas Clarke y Park, ampliamente utilizadas en sistemas de generación se señales alternas.**

En este trabajo se sintetiza uno de estos algoritmos usando el método de síntesis de alto nivel brindado por AMD Xilinx mediante una de sus herramientas de desarrollo como lo es Vitis. Para ello se toma un diseño realizado en diagrama de bloques y se lo traduce código Cpp para su síntesis.

***Index Terms*—SOGI, PLL, FPGA, HLS.**

I. Introducción

SON muchos los algoritmos de detección de fase que se utilizan en la inyección de energía a la red eléctrica. Estos algoritmos pueden ser implementados en código C++ u otros lenguajes de programación para ser ejecutados en algún microprocesador, obteniéndose las señales necesarias para el acoplamiento de algún sistema generador de energía eléctrica con la red.

Algunos microprocesadores que se utilizan para implementar estos algoritmos son los de la familia C2000 Delfino de Texas Instrument [1] que poseen módulo de operaciones con punto flotante en hardware, como así salida PWM, conversor analógico-digital (ADC) por encima de los 12.5 mega muestras por segundo (MSPS) e instrucciones de funciones trigonométricas de 1 a tres ciclos lo que los hace ideales para este tipo de sistemas.

Pero, en la actualidad, las FPGA presentan mejores características que les permiten ser más eficientes a la hora de utilizarlas para el diseño de estos sistemas. El hardware mismo pasa a ser más maleable a la hora de encontrar un diseño óptimo.

En este trabajo se realiza el diseño de un módulo hardware de un detector de fase usando el modelo SOGI PLL SRF. Para ello primero se traduce a código C el diagrama de bloques, y usando ese código, se genera el módulo usando la síntesis de alto nivel o HSL mediante la herramienta Vitis de AMD Xilinx [2].

II. Trabajos relacionados

EN la actualidad hay muchos trabajos que abordan los algoritmos de sincronización y acoplamiento de energía a redes eléctricas, siendo este un tema muy activo.

En [3] el autor hace una comparativa de distintos algoritmos de sincronización tanto monofásicos como trifásicos. Para la simulación usa Simulink, de Matlab con los cuales logra tener la respuesta del algoritmo a distintos tipos de fallas. También logra ejecutar el algoritmo en el DSP TMS320F28335 de Texas Instrument obteniendo las respuestas esperadas.

De la misma manera, en [4] el autor plantea los problemas que representan la importancia de la sincronización de señales de sistemas de comunicación, eliminación de ruidos o atenuación, retrasos a los cuales se busca soluciones eficientes mediante la utilización de PLL, realizando una comparativa entre dos de sus tipologías mediante simulación en Matlab y Simulink.

En [5] el autor propone la elección de un algoritmo para la estimación de ángulo de fase en función del desempeño demostrado frente a distintas perturbaciones típicas de la red de distribución para su implementación, basado en la técnica Phase Locked-Loop (PLL).

En [6] los autores plantean el desarrollo de algoritmos de sincronización que trabajen en conjunto con sistemas de electrónica de potencia y sistemas de control, con la finalidad de lograr que dos o más fuentes externas estén sincronizadas entre sí y evitar daños en dispositivos finales; tomando como objeto de estudio aquellos que son lazo de bloque de fase.

En [7] los autores proponen la realización del modelo dinámico en base a una representación a pequeña señal de un inversor trifásico en un marco de referencia dq y su posterior prueba en simulación de un arreglo fotovoltaico conectado a la red con el fin de entregar la máxima potencia disponible a la red eléctrica. Esto implico el diseño de un PLL digital para establecer un referencia de sincronismo y así caracterizar el sistema con dos lazos de control para regular tanto el voltaje de entrada y la corriente de salida.

En [8] el autor realiza el análisis del algoritmo de sincronización p-PLL basado en la teoría de las potencias instantáneas de Akagi inicialmente por medio de un estudio de las principales características de los métodos de sincronización existentes y su utilización en sistemas trifásicos con el objetivo de detectar correctamente el ángulo de fase de la componente fundamental. Así, posteriormente, profundiza en detalles respecto a los procesadores de señales digitales enfocándose en el módulo de evaluación ICETEK-LF2407-C el cual representa la plataforma en la cual desarrolla el algoritmo p-PLL. Concluye, así en una explicación sobre la construcción del hardware y el software, así como los resultados obtenidos en las simulaciones y experimentos llevados a cabo.

Como se ha podido observar, en los trabajos analizados se describe la necesidad de implementar la técnica Phase Locked-Loop (PLL) para establecer una referencia de sincronismo y lograr la sincronización entre una o más fuentes externas. Adicionalmente, los trabajos presentados implementan dichos algoritmos en lenguaje c y no en VHDL, por lo que no es posible observar su funcionamiento en placas de desarrollo que posean FPGA.

En este trabajo se propone traducir el diagrama de bloques de un detector de fase como lo es el SOGI PLL SRF a código C++ y sintetizarlo en lenguaje de descripción de hardware usando la técnica HLS.

III. Detector de fase SOGI PLL SRF

EL diagrama de bloques del algoritmo SOGI PLL SRF se muestra en la figura 1.

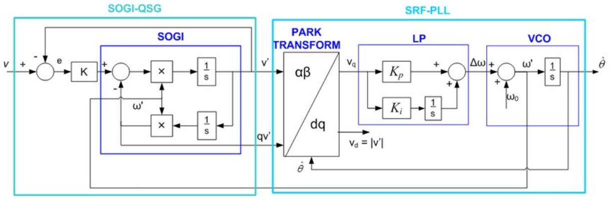


Fig. 1. IP Generador de Fallas.

En el diagrama de bloques se pueden observar las partes que conforman el algoritmo. La señal de entrada ingresa al bloque SOGI, el cual se conecta con el bloque de la transformada PARK. Luego sigue un controlador PI para finalizar en un integrador. La salida de este integrador es la fase detectada de la señal de entrada.

A. *Generación de señal en cuadratura con un SOGI*

Calculando las funciones de transferencias que entrega el diagrama del SOGI, estas quedan como se ven en las ecuaciones 1 y 2.

$$G(s) = \frac{v'(s)}{v(s)} = \frac{k\omega' s}{s^2 + k\omega' s + \omega'^2} \tag{1}$$

$$Gq(s) = \frac{qv'(s)}{v(s)} = \frac{\omega'^2}{s^2 + k\omega' s + \omega'^2} \tag{2}$$

Para poder implementar el SOGI en una FPGA se deben discretizar las ecuaciones haciendo:

$$s = \frac{2}{T}\frac{z-1}{z+1} \tag{3}$$

Reemplazando, el SOGI discretizado queda:

$$H(z) = \frac{v'(z)}{v(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 - a_1 z^{-1} - a_2 z^{-2}} \tag{4}$$

$$Hq(z) = \frac{v'(z)}{v(z)} = \frac{c_0 + c_1 z^{-1} + c_2 z^{-2}}{a_0 - a_1 z^{-1} - a_2 z^{-2}} \tag{5}$$

donde

$$a_0 = 1 \tag{6}$$

$$a_1 = \frac{2(4 - \omega'^2 T_s^2)}{2k\omega' T_s + \omega'^2 T_s^2 + 4} \tag{7}$$

$$a_2 = \frac{2k\omega' T_s - \omega'^2 T_s^2 - 4}{2k\omega' T_s + \omega'^2 T_s^2 + 4} \tag{8}$$

$$b_0 = \frac{2k\omega' T_s}{2k\omega' T_s + \omega'^2 T_s^2 + 4} \tag{9}$$

$$b_1 = 0 \tag{10}$$

$$b_2 = -b_0 \tag{11}$$

$$c_0 = \frac{\omega'^2 T_s^2}{2k\omega' T_s + \omega'^2 T_s^2 + 4} \tag{12}$$

$$b_1 = 2C_0 \tag{13}$$

$$b_2 = c_0 \tag{14}$$

Para la codificación en C++ el SOGI queda

```
v'[0] = a1*v'[1] + a2*v'[2] +
                  b0*(v[0] - v[2]);

qv'[0] = a1*qv'[1] + a2*qv'[2] +
          k*c0*(v[0] + 2*v[1] + v[2]);
```

Aplicando los mismos criterios para la transformada Park, el controlador PI y para el integrador, la codificación en C++ queda:

```
//Transformada park
vq[0] = v'[0]*cos(tita[0]) +
                 qv'[0]*sin(tita[0]);
```

```
//Controlador PI
pi_out[0] = pi_out[1] + (kp +
    Ki*Ts/2)vq[0] + (ki*Ts/2-kp)*vq[1];

//Frecuencia de red
omega[0] = Wred + pi_out[0];

//Fase de red [integrador]
tita[0] = tita[1] + Ts*omega[1];
```

donde Ts es el periodo entre muestras, Wred es la velocidad angular de la señal de red alterna, Kp y Ki son las constantes del controlador PI y tita es el ángulo de fase detectado de la señal de red de entrada.

B. Codificación del SOGI PLL SRF en c

Con lo obtenido en los párrafos anteriores se obtiene el código en C++ del SOGI PLL SRF. Antes se reemplazan v' por Vd_sogi y qv' por Vq_sogi.

```
void sogi_pll_srf(
float Vred, //entrada señal de red
float *frecuencia, //salida frecuencia
float *fase, //salida fase
float *Vacoplada) //salida señal en fase
{

V[0] = Vred;
//Calcula salida del SOGI
Vd_sogi[0] = a1*Vd_sogi[1] + a2*Vd_sogi[2]
                       + b0*(V[0] - V[2]);
Vq_sogi[0] = a1*Vq_sogi[1] + a2*Vq_sogi[2]
            + k*c0*(V[0] + 2*V[1] + V[2]);

//Transformada park
Vpark_q[0] = Vd_sogi[0]*cos(tita[0])
               + Vq_sogi[0]*sin(tita[0]);

//controlador PI
pi_out[0] = pi_out[1] + (kp + Ki*Ts/2)
    Vpark_q[0] + (ki*Ts/2-kp)*Vpark_q[1];

//velocidad de fase de red
omega[0] = Wred + pi_out[0];

//Fase de red [integrador]
tita[0] = tita[1] + Ts*omega[1];

if(tita[0] > (2*pi)){
   tita[0] -= 2*pi;
   pi_out[0] = 0;
}

//salidas
*frecuencia = omega[0]/(2*pi);
*fase = tita[0];
*Vacoplada = Vd_sogi[0];

//actualiza arreglos
V[2] = V[1];
V[1] = V[0];
Vd_sogi[2] = Vd_sogi[1];
Vd_sogi[1] = Vd_sogi[0];
Vq_sogi[2] = Vq_sogi[1];
Vq_sogi[1] = Vq_sogi[0];
Vpark_q[1] = Vpark_q[0];
pi_out[1] = pi_out[0];
omega[1] = omega[0];
tita[1] = tita[0];
}
```

Para la prueba se diseñó otro código, el cual genera algunas de las fallas típicas en la red eléctrica.

```
int main (int argc, char **argv) {
  FILE   *fp;
  float Vred, frecuencia,
     fase, Vacoplada,t=0;

  fp=fopen("salida.txt","w");

  //se genera señal con fallas
  for(int i=0;i<16000;i++) {
    if(i<4000){
      //señal pura
      ug = Vg*sqrt(2)*sin(w_net*t);
      t += T_sample;
    }else if(i>= 4000 && i<8000){
      //señal con un hueco
      ug = Vg*sqrt(2)*(1-0.4)*
                      sin(w_net*t);
      t += T_sample;
    }else if(i>= 8000 & i<12000){
      //señal con salto de fase
      ug = Vg*sqrt(2)*
             sin(w_net*t+(pi/(6)));
      t += T_sample;
    }else if(i>= 12000 & i<16000){
      //señal con cambio de frecuencia
      ug = Vg*sqrt(2)*
          sin((w_net+2*pi)*t);
      t += T_sample;
    }
    //se llama a la función
    //sogi_pll_srf
    sogi_pll_srf(Vred, &frecuencia,
                    &fase, &Vacoplada);

    //los resultados se guardan en un
    //archivo de texto
    fprintf(fp,"%1.4f   %1.4f   %1.4f
              %1.4f\n", Vred, frecuencia,
                        fase, Vacoplada);
  }
```

```
    fclose(fp);
    return 0;
}
```

IV. Síntesis del algoritmo SOGI PLL SRF

Para la síntesis del algoritmo se utilizó la herramienta Vitis hls de Xilinx AMD. Esta herramienta permite cargar los código en c o c++ de la función a sintetizar junto con el archivo, también escrito en c o c++, para realizar el test.

Para cumplir con el diseño se deben cumplir cuatro pasos.

- **Primer paso**: se debe probar el módulo a sintetizar con algún compilador. Puede ser cualquiera que compile código c o c++.
- **Segundo paso**: una vez probado el código se debe sintetizar para y orientar la síntesis con algunas directivas que pueden mejorar la respuesta o el uso de recursos del hardware generado.
- **Tercer paso**: una vez obtenida la síntesis se debe realizar la cosimulación, es decir, probar el hardware generado mediante la síntesis.
- **Cuarto paso**: si la cosimulación fue correcta se genera el módulo hardware.

En este trabajo se realizaron tres códigos para generar el módulo hardware. En el primero se utilizaron variables del tipo **float**, el segundo se utilizaron variables del tipo **double** y el tercero se utilizaron variables del tipo **punto fijo**.

A. Simulaciones con compilador c

En estas simulaciones se utilizó el compilador Gcc de linux. Los resultados se pueden observar en las figuras

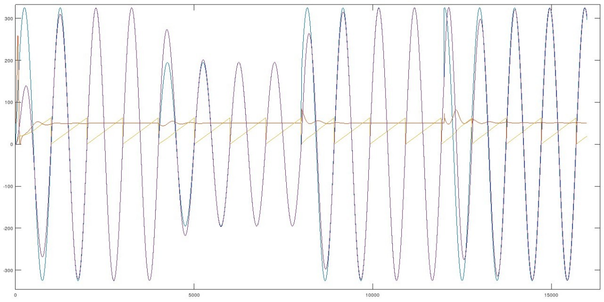


Fig. 2. Simulación con variable tipo float usando compilador Gcc.

En las figuras 2, 3, 4 aparecen cuatro señales:

- En celeste la señal de entrada o prueba que está formada por cuatro ciclos de señal sin falla, cuatro ciclos con falla de hueco de tensión, cuatro ciclos con salto de fase y cuatro ciclos con variación de frecuencia.
- En azul la señal alterna en fase generada por el SOGI
- En rojo el valor de la frecuencia. Es la señal que muestra las perturbaciones del sistema al producirse la falla
- En naranja la señal de la fase detectada por el SOGI PLL SRF. En la gráfica la señal está multiplicada por 10 para que se pueda apreciar al lado de las otras señales.

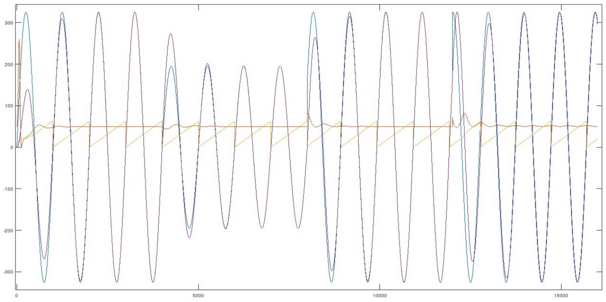


Fig. 3. Simulación con variable tipo double usando compilador Gcc.

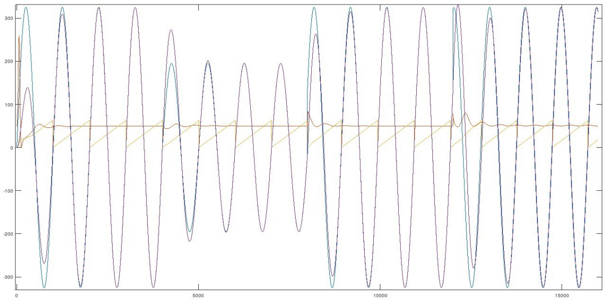


Fig. 4. Simulación con variable tipo fixed usando compilador Gcc.

B. Síntesis del módulo SOGI PLL SRF

La síntesis se ha realizado sin poner directivas. El diseño se hizo para una frecuencia de clock de 100MHz, para la placa ZedBoard Zynq Evaluation Developement Kit de Digilent [10] basada en un SoC XC7Z020CLG484-1 de la serie Zynq 7000.

Los resultados de la síntesis se pueden observar en la Tabla I. En ella se puede observar la diferencia en recursos al usar variables tipo flotantes y variables tipo doble. En el caso del uso de variables tipo punto fijo se utilizó una variable de 48 bits con 24 bits para la parte decimal.

TABLE I
Recursos usados para distintos tipos de variables

Variable	BRAM	DSP	FF	LUT	URAM
Float	-	28	4986	8469	-
Double	16	98	15797	19141	-
Fixed	-	24	17784	26689	

En la tabla II se pueden ver las latencias o cuantos ciclos de reloj tarda cada diseño en devover un valor calculado.

TABLE II
Latencia de cada módulo según el tipo de variable

Variable	LATENCIA (NS)	LATENCIA (CICLOS)
Float	930	93
Double	1.840	184
Fixed	820	82

C. Cosimulación del módulo en sus distintas variantes

La cosimulación del módulo diseñado en los tres tipos de variables coincidió con la simulación usando el compilador Gcc, la cual se puede observar en las figuras 5, 6 y 7.

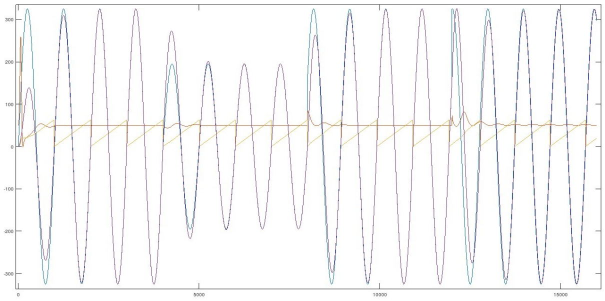


Fig. 5. Co-simulación con variable tipo float.

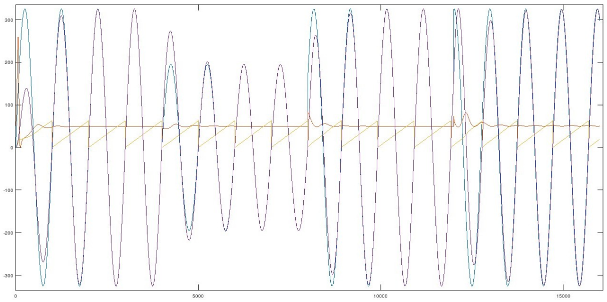


Fig. 6. Co-simulación con variable tipo double.

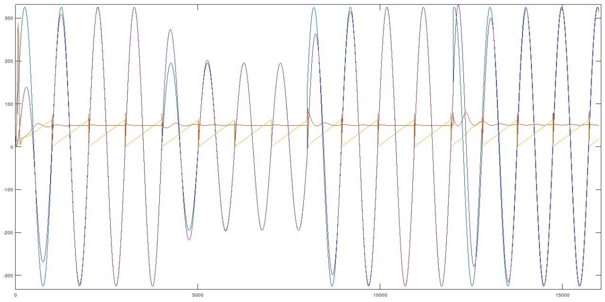


Fig. 7. Co-simulación con variable tipo fixed.

D. Generación del IP

En las figuras 8, 9 y 10 se pueden observar los IPs generados.

Los puertos generado son:

- ap_start, ap_done, ap_idle y ap_ready: puertos de control generados automáticamente. De estos puertos se utiliza el puerto ap_start para ir generado los valores que hacen a la señal alterna de salida.
- ap_clk y ap_rst_n: reloj y reset.
- Vred: señal de entrada alterna de la red.
- frecuencia_ap_vld: validación de la señal frecuencia.
- fase_ap_vld: validación de la señal fase.
- Vacoplada_ap_vld: validación de la señal Vacoplada.
- frecuencia: puerto por donde sale la señal frecuencia.
- fase: puerto por donde sale la señal fase.
- Vacoplada: puerto por donde sale la señal Vacoplada.

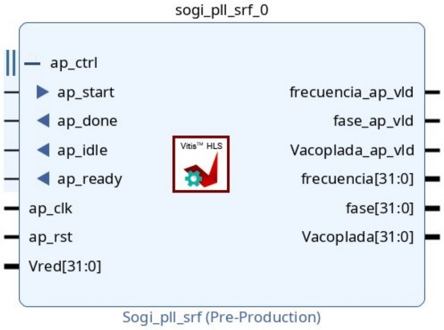


Fig. 8. IP SOGI PLL SRF diseñado con variables tipo float

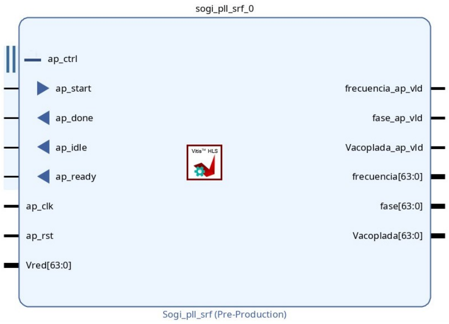


Fig. 9. IP SOGI PLL SRF diseñado con variables tipo double

V. Conclusiones

Este trabajo ha permitido realizar un diseño de un bloque hardware de un seguidor de fase el cual se ha realizado de forma sencilla. Esto permite que en poco tiempo se tenga un módulo harware funcional y de rápida implementación.

En el caso de los puertos de entrada y salida se ha dejado que sean del ancho del tipo de variable que se estaba utilizando. Esto se puede modificar en el diseño.

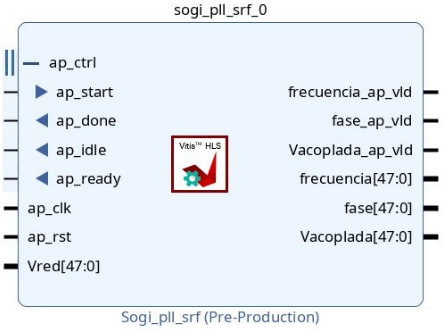


Fig. 10. IP SOGI PLL SRF diseñado con variables tipo fixed

REFERENCES

[1] Delfino™ Premium Performance mcus (2020) TI Training. Available at: https://training.ti.com/delfino-premium-performance-mcus (Accessed: December 14, 2022).

[2] Vitis software platform (no date) Xilinx. Available at: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (Accessed: December 14, 2022).

[3] Daniel Serrano Dominguez. Análisis comparativo de téctnicas de sincronización con la red eléctrica. Trabajo Final de Carrera. Universidad de Sevilla. Julio 2014.

[4] Cynthia Manosalvas Pillajo. Diseño e Implementación de un lazo de enganche de fase (PLL) en un microcontrolador. Trabajo Final de Carrera. Universidad de Zaragoza. 2019.

[5] Algoritmos de detección de fase para sincronización y control de frecuencia de central micro hidráulica plug & play. Trabajo final de carrera. Carlos Patricio Aedo Paredes. Universidad de Chile. Julio 2014

[6] Andrés Antonio Segovia Vásquez. Implementación y análisis de algoritmos de sincronización de fase, para fuentes de energía renovables en sistemas trifásicos. Trabajo de titulación previo a la obtención del título de Ingeniero en Electrónica y Telecomunicaciones. Universidad de Cuenca. Julio 2021.

[7] César Augusto Prieto Suárez; Juan David Peña Alvarado. Diseño y Simulación de las Diferentes Etapas de Control en una Micro red Eléctrica. Proyecto Curricular de Ingeniería Electrónica. Universidad Distrital Francisco José de Caldas. 2018.

[8] Alejandro Méndez Samper. Implementación y simulación del algoritmo de sincronización p-PLL mediante un DSP LF2407A. Trabajo de Diploma para optar por el título de Ingeniero Electricista. Universidad Tecnológica de La Habana. Junio 2016.

[9] Antonella Nagliero, Rosa Mastromauro, Marco Liserre, Antonio Dell'Aquila. (2010). Monitoring and synchronization techniques for single-phase PV systems. 1404 - 1409. 10.1109/SPEEDAM.2010.5545057.

[10] ZedBoard - Digilent Reference. Available at: https://digilent.com/reference/programmable-logic/zedboard/start (Accessed: December 14, 2022).

Generador de fallas para pruebas de algoritmos de sincronización con la red eléctrica monofásica

Alejandro Núñez Manquez
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
janyo12@unsl.edu.ar

Julio Dondo Gazzano
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
jdondo@gmail.com

Estrella Gómez Orozco
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
estrellagomez08@gmail.com

Carlos F. Sosa Paez
Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
San Luis, Argentina
sosapaez@gmail.com

***Abstract*—Uno de los campos de estudio actuales es el acoplamiento entre sistemas generadores de energía eléctrica alterna. En este tipo de acoplamiento es necesaria la detección de unos atributos de la tensión de la red a la cual se desea acoplar, los cuales son el ángulo de fase y la amplitud de la señal o señales de tensión según el sistema el cual puede ser monofásico o trifásico.**

En este trabajo se diseñará un banco de prueba de algoritmos de sincronización implementado en FPGA. Este banco de prueba permite configurar distintas fallas en la red como por ejemplo huecos, salto de fase, salto de frecuencia y aparición de armónicos. Los algoritmos a probar se implementan en lenguaje c y mediante la técnica HLS (Higth Level Syntesis) se convierte a VHDL para su implementación en una FPGA.

***Index Terms*—SOGI, PLL, FPGA, HLS.**

I. Introducción

La producción de la energía eléctrica siempre ha sido un tema central en el desarrollo de cada país. Con el avance de la tecnología se ha logrado que la producción de la misma esté más al alcance del público en general permitiendo que se pueda sustentar una vivienda con la producción propia de energía, la cual puede ser solar, eólica, etc. Esto a generado sistemas híbridos los cuales pueden usar tanto la energía proveniente de la red eléctrica implementada por las empresas productoras y/o distribuidoras de la misma como por la propia producción de energía del usuario.

En el caso de que un usuario genere energía que no la pueda acopiar usando baterías, esa energía se puede inyectar a la red eléctrica para lo cual se debe acoplar a la misma usando alguna técnica de acoplamiento.

En la actualidad hoy muchas técnicas de acoplamiento de energía, tanto para sistemas monofásico como para sistemas trifásicos. Estas técnicas detectan el ángulo de fase de la red como su amplitud, datos necesarios para el acoplamiento. También estos datos deben estar acoplados en un tiempo menor al periodo de la red electrica a la cual se desea acoplar. Para ello se han diseñado muchos algoritmos que logran estos objetivos.

En este trabajo se va a sintetizar en VHDL un banco de pruebas para generar tipos de fallas propias de una red eléctrica. Junto a la síntesis del generador de fallas se sintetiza uno de los tantos algoritmos de detección de fase, en este trabajo se utiliza el algoritmo SOGI PLL SRF.

II. Trabajos relacionados

Son muchos los trabajos que abordan los algoritmos de sincronización y acoplamiento de energía a redes eléctricas.

En [1] el autor hace una comparativa de distintos algoritmos de sincronización tanto monofásicos como trifásicos. Para la simulación usa Simulink, de Matlab con los cuales logra tener la respuesta del algoritmo a distintos tipos de fallas. También logra ejecutar el algoritmo en el DSP TMS320F28335 de Texas Instrument obteniendo las respuestas esperadas.

De la misma manera, en [2] el autor plantea los problemas que representan la importancia de la sincronización de señales de sistemas de comunicación, eliminación de ruidos o atenuación, retrasos a los cuales se busca soluciones eficientes mediante la utilización de PLL, realizando una comparativa entre dos de sus tipologías mediante simulación en Matlab y Simulink.

En [3] el autor propone la elección de un algoritmo para la estimación de ángulo de fase en función del desempeño demostrado frente a distintas perturbaciones típicas de la red de distribución para su implementación, basado en la técnica Phase Locked-Loop (PLL).

En [4] los autores plantean el desarrollo de algoritmos de sincronización que trabajen en conjunto con sistemas de electrónica de potencia y sistemas de control, con la finalidad de lograr que dos o más fuentes externas estén sincronizadas entre sí y evitar daños en dispositivos finales; tomando como objeto de estudio aquellos que son lazo de bloque de fase.

En [5] los autores proponen la realización del modelo dinámico en base a una representación a pequeña señal de un inversor trifásico en un marco de referencia dq y su posterior prueba en simulación de un arreglo fotovoltaico conectado a la red con el fin de entregar la máxima potencia disponible a la red eléctrica. Esto implico el diseño de un PLL digital para establecer un referencia de sincronismo y así caracterizar el sistema con dos lazos de control para regular tanto el voltaje de entrada y la corriente de salida.

En [6] el autor realiza el análisis del algoritmo de sincronización p-PLL basado en la teoría de las potencias instantáneas de Akagi inicialmente por medio de un estudio de las principales características de los métodos de sincronización existentes y su utilización en sistemas trifásicos con el objetivo de detectar correctamente el ángulo de fase de la componente fundamental. Así, posteriormente, profundiza en detalles respecto a los procesadores de señales digitales enfocándose en el módulo de evaluación ICETEK-LF2407-C el cual representa la plataforma en la cual desarrolla el algoritmo p-PLL. Concluye, así en una explicación sobre la construcción del hardware y el software, así como los resultados obtenidos en las simulaciones y experimentos llevados a cabo.

Como se ha podido observar, en los trabajos analizados se describe la necesidad de implementar la técnica Phase Locked-Loop (PLL) para establecer una referencia de sincronismo y lograr la sincronización entre una o más fuentes externas. Sin embargo, estos algoritmos de sincronización no siempre cuentan con la versatilidad de configuración que permite implementar distintas fallas en la red en un mismo banco de prueba. Adicionalmente, los trabajos presentados implementan dichos algoritmos en lenguaje c y no en VHDL, por lo que no es posible observar su funcionamiento en placas FPGA.

En este trabajo se propone un banco de prueba reconfigurable que toma en cuenta todas las fallas en la red (huecos, salto de fase, salto de frecuenncia y aparición de armónicos) y que además de implementarse en c, mediante la técnica HLS es fácilmente convertible a VHDL para su posterior implementación en FPGA.

III. Generador de fallas

PARA la prueba de los algoritmos de detección de fase es necesario inyectarles las señales típicas de fallas en redes eléctricas para verificar su desempeño. Estas fallas pueden ser:

- Huecos de tensión: es una disminución brusca de la tensión de alimentación a un valor situado entre el 90% y el 1% de la tensión declarada, seguida de un restablecimiento de la tensión después de un corto lapso de tiempo. Por convenio un hueco dura entre 10 ms a 1 minuto. El generador debe generar estos huecos de tensión, aunque la duración de los mismos se fijará en 5 ciclos de la frecuencia de la red.
- Variación de frecuencia: dependiendo del lugar de donde sea la red eléctrica, la frecuencia nominal puede ser de 50 Hz o de 60 Hz. No obstante esta frecuencia puede variar dependiente de los desequilibrios que se produzcan entre la generación y la carga. Para las pruebas el generador debe poder generar perturbación de este tipo con variaciones de frecuencia que no superen el 10% de la frecuencia nominal.
- Variación de fase: es un salto abrupto de la fase del sistema generando con ello una falla. El generador debe poder generar un salto de fase de hasta $\frac{\pi}{4}rad$
- Armónicos: la tensión armónica es una tensión que se suma a la tensión nominal con una frecuencia múltiplo de la frecuencia nominal. Generalmente los armónicos que se encuentran y que más destacan en la red eléctrica son los terceros, quintos y séptimos. El generador debe poder generar armónicos con un módulo de hasta el 10% de la tensión de la fundamental.

De acuerdo a las fallas estipuladas en las líneas de transmisión se diseña el código del generador de fallas que permite, mediante el cambio de estado de algunos parámetros, generar tantas fallas en simultáneo como se desee, así como la intensidad de las mismas.

A. Diseño del generador de fallas con HLS

Para el diseño de este bloque se utilizó un código programado en C y sintetizado en Vivado-HLS. Dicho código posee seis entradas correspondientes a las fallas indicadas en el párrafo anterior. Además se incluye una entrada de dos bits para controlar la variación de fase, de frecuencia y el módulo de los huecos de tensión. El control se diseña de esta forma para utilizar los recursos que posee la placa en donde se implementa el sistema.

1) Generación de la falla hueco de tensión: Para implementar esta falla se la debe habilitar desde el módulo control. Además se debe seleccionar con qué módulo se activa la falla. En el código se puede observar la configuración de la misma.

```
if(falla_hueco == 0){
   tensionSalida = 1;
}else{
   if(modulo==0){
      tensionSalida = 0.9;
   }else if(modulo==1){
      tensionSalida = 0.8;
   }else if(modulo==2){
      tensionSalida = 0.6;
   }else{
      tensionSalida = 0.4;
   }
}
```

En la Figura 1 se observa la salida de la señal generada.

2) Generación de la falla por variación de frecuencia: Asimismo, en el generador de fallas se ve contemplada la variación de frecuencia como otra de las posibles fallas. En el código se puede observar la configuración de la misma.

```
if(falla_frecuencia == 0){
   delta_frecuencia = 0;
}else{
   if(modulo==0){
```

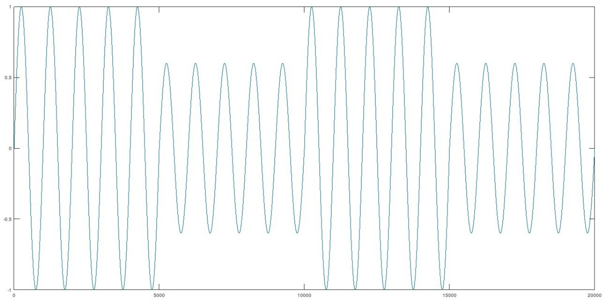


Fig. 1. Falla hueco con un valor del 60% del valor nominal.

```
      delta_frecuencia = 1%;
   }else if(modulo==1){
      delta_frecuencia = 3%;
   }else if(modulo==2){
      delta_frecuencia = -1%;
   }else{
      delta_frecuencia = -3%;
   }
}
```

En la Figura 2 se observa la salida de la señal generada.

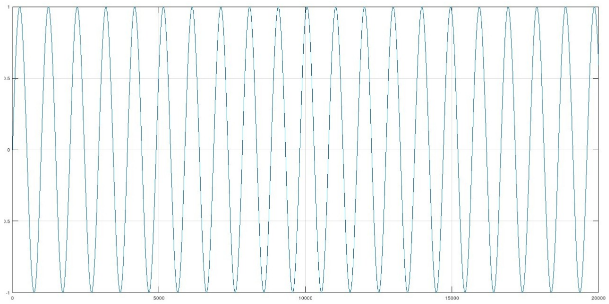


Fig. 2. Falla de frecuencia con un cambio del 1% menos del valor nominal

3) Generación de la falla por salto de fase: En cuanto a la variación de fase, el procedimiento es similar al descrito previamente.

```
if(falla_fase == 0){
   delta_fase = 0;
}else{
   if(modulo==0){
      delta_fase = pi/6;
   }else if(modulo==1){
      delta_fase = pi/4;
   }else if(modulo==2){
      delta_fase = pi/3;
   }else{
      delta_fase = pi/2;
   }
}
```

En la Figura 3 se observa la salida de la señal generada.

4) Generación de la falla por armónicos: La configuración de la falla por armónicos se muestra en el siguiente código:

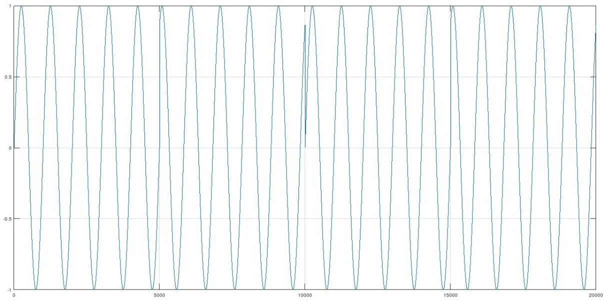


Fig. 3. Falla de fase con un salto de 60 grados.

```
if(f_arm3 == 0){
   mod_arm3 = 0;
}else{
   mod_arm3 = 0.1*tensionSalida;
}

if(f_arm5 == 0){
   mod_arm5 = 0;
}else{
   mod_arm5 = 0.1*tensionSalida;
}

if(f_arm7 == 0){
   mod_arm7 = 0;
}else{
   mod_arm7 = 0.1*tensionSalida;
}
```

En la Figura 4 se observa la salida de la señal sumados los tres armónicos generados.

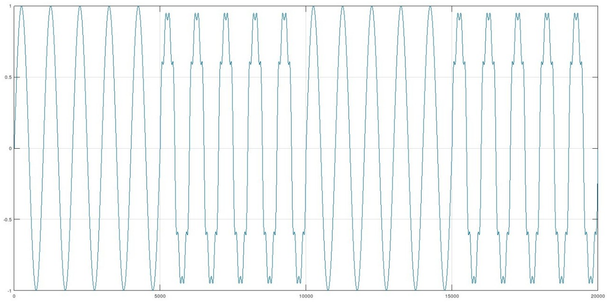


Fig. 4. Falla por inyección de armónicos.

El código de salida del generador de falla se muestra a continuacción

```
if(cont <5000){
fase_a = sinf(w*t);
}else{
fase_a = V_mod*sinf(w*t + delta_fase) + mod_3*sinf
}
```

B. Generación del IP

En la figura 5 se puede observar el IP generado. El mismo posee conexión con el Bus AXI para el control de la inyección de fallas.

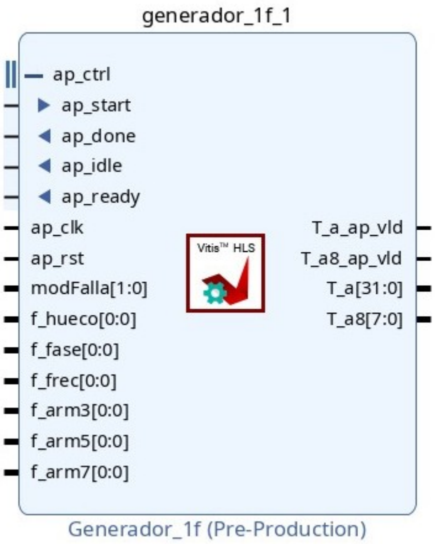


Fig. 5. IP Generador de Fallas.

Los puertos generado son:

- ap_start, ap_done, ap_idle y ap_ready: puertos de control generados automáticamente. De estos puertos se utiliza el puerto ap_start para ir generado los valores que hacen a la señal alterna de salida.
- modFalla: puerto desde donde se configura el valor de la falla para falla por hueco de tensión, falla por variación de frecuencia y falla por salto de fase.
- f_hueco: puerto para habilitar o deshabilitar la falla por hueco de tensión.
- f_fase: puerto para habilitar o deshabilitar la falla por salto de fase.
- f_frec: puerto para habilitar o deshabilitar la falla variación de frecuencia.
- f_arm3, f_arm5 y f_arm7: puertos para habilitar o deshabilitar las fallas por componentes armónicas.
- ap_clk y ap_rst_n: reloj y reset.
- T_a: puesto por donde sale la señal generada.
- T_a_ap_vld: control de validación de la señal de salida.

IV. Detector de fase Sogi PLL SRF

PARA la prueba del banco se sintetizó en HLS el detector de fase SOGI PLL SRF, que se muestra en la figura 6.

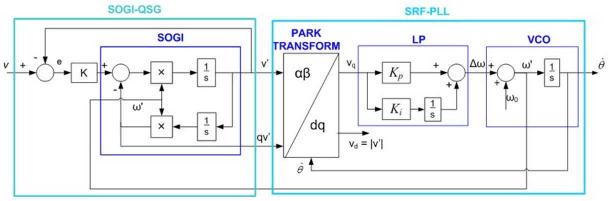


Fig. 6. IP Generador de Fallas.

Para ello se tuvo que convertir a código C el diagrama de bloques, el cual se probó para ver su respuesta, la cual se muestra en la figura 7.

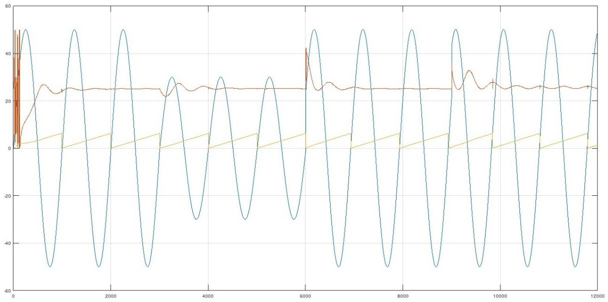


Fig. 7. Respuesta del SOGI PLL SRF ante una señal de entrada con falla de hueco de tensión.

En esta gráfica se pueden observar las siguientes señales:

- En azul: señal alterna de entrada con una falla por hueco de tensión.
- En rojo: señal de salida correspondiente a la frecuencia de la señal de entrada, en este caso 50 Hz.
- En naranja: señal de salida correspondiente a la fase de la señal de entrada.

obteniéndose el módulo que se observa en la figura 8.

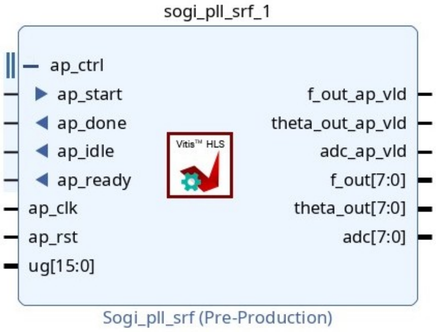


Fig. 8. IP Generador de Fallas.

Los puertos generado son:

- ap_start, ap_done, ap_idle y ap_ready: puertos de control generados automáticamente. De estos puertos se utiliza el puerto ap_start para ir generado los valores que hacen a la señal alterna de salida.
- ug: puerto de entrada de 16 bits.
- ap_clk y ap_rst_n: reloj y reset.
- f_out: salida del valor de frecuencia de la señal
- theta_out: Fase de la señal alterna.
- adc: Señal de entrada al módulo que se saca como salida para la comparación con la señal theta_out.

V. Sistema propuesto

EL sistema propuesto consta de un generador de fallas y un detector de fase implementando el algoritmo SOGI PLL SRF. En la figura 9 se pueden observar los bloques que son parte del sistema.

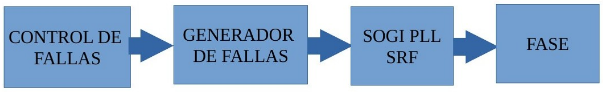


Fig. 9. Diagrama de bloques del sistema completo.

Para poder generar la señal alterna la señal de salida del generador de fallas se conecta con un módulo PWM que toma esa señal y genera una alterna PWM. Esa señal alterna es filtrada mediante un filtro RC con un frecuencia de corte de 400 Hz.

La señal alterna es capturada por un conversor analógico digital e ingresada al módulo SOGI PLL SRF. Las señales que estrega este módulo se conectan a un conversor digital analógico.

A. Diseño del hardware del sistema propuesto

La implementación del generador de fallas antes mencionado no podría existir sin un sistema que lo complemente y que le permita cumplir su tarea. En la Figura 10 se observa el Diagrama de bloques propuesto, en el cual se distinguen tres bloques principales sobre los que se profundiza: el bloque Generador de fallas, el bloque PWM y el bloque SOGI-PLL.

Además de los bloques antes mencionados, se incluyen un bloque ctrl_switch y ctrl_leds; el primero permite habilitar o deshabilitar las fallas deseadas y el segundo indica cuál es la configuración del bloque Generador de fallas en cada momento. También se incluyen un bloque PmodADC y un bloque PmodDAC.

- **Generador de fallas**: este bloque genera una señal monofásica controlada. Su salida es la señal Ta que alimenta al bloque PWM, cuyo rango es desde -1 a 1.
- **PWM**: Este bloque tiene como entrada la señal generada por el bloque **Generador de fallas** y genera la señal PWM correspondiente. También genera el pulso de inicio (ap_start) del Generador de fallas y del SOGI PLL SRF. La señal PWM de salida alimenta un filtro RC pasa bajas con una frecuencia de corte de 400 Hz.
- **PmodADC**: este bloque digitaliza la salida del filtro RC. Su salida alimenta al bloque SOGI PLL SRF.
- **SOGI PLL SRF**: Este bloque tiene como entrada la señal generada por el bloque PmodADC. Tiene como salidas corresponden la fase capturada de la señal alterna, el valor de la frecuencia de esa señal y la señal ADC de entrada a este bloque para control. Estas señales son ingresadas al bloque PmodDAC.
- **PmodDAC**: este bloque permite mostrar por osciloscopio las señales generadas por el bloque SOGI PLL SRF.

VI. CONFIGURACIÓN PROPUESTA

LA placa de desarrollo utilizada en este trabajo es la ZedBoard Zynq Evaluation Developement Kit [6] de Digilent basada en un SoC XC7Z020CLG484-1 de la serie Zynq 7000. La placa está equipada con osciladores de 33,333MHz para el PS y de 100MHz para la PL, 5 conectores para módulos Digilent Pmod™ de 2x6 pines (1 conectado al PS y 4 conectados al PL), 2 botones de reinicio (1 para PS y 1 para PL), 7 pulsadores (2 para PS y 5 para PL), 8 interruptores (PL) y 9 LED (1 para PS y 8 para PL), interfaces USB-JTAG para programación y depuración. El kit se muestra en la figura 11.

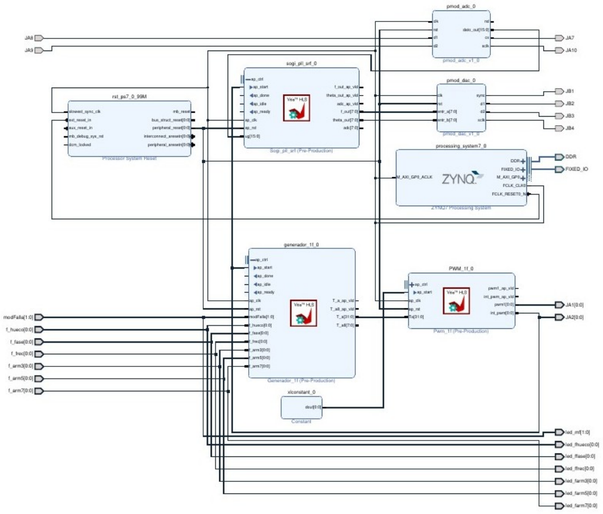


Fig. 10. Sistema propuesto diseñado en Vivado.

La PL del SoC posee un Xilinx Artix®-7 FPGA que contiene 85000 celdas lógicas, 53200 LUT, 106400 FF, 560KB de bloques RAM extensibles y 220 bloques DSP programables con sumadores/acumuladores de 48 bits, multiplicadores con signo de 18x25, y pre-sumadores de 25 bits capaces de operar a 741MHz con un desempeño pico para un FIR simétrico de 276GMAC.

El PS del SoC se basa en un dual-core ARM® Cortex™-A9 de 2.5DMIPS/MHz por CPU, arquitectura ARMv7-A, NEON™ media-processing engine, Vector Floating Point Unit (VFPU) de simple y doble precisión, temporizadores, varias cachés, memorias on-chip ROM de booteo, 256KB de RAM (OCM), UART de hasta 1Mb/s e I2C M/S), alto BW de conectividad entre PS y PL, ARM AMBA® basado en AXI, etc. Para la versión -1 del XC7Z020, el reloj del CPU puede operar hasta 667MHz.

VII. PRUEBAS Y RESULTADOS.

LOS Los módulos **Generador de Fallas** y **SOGI PLL SRF** fueron probados de forma individual durante el proceso de diseño usando la herramienta vitis_hls de Xilinx, cuyas respuestas han sido mostradas en las secciones anteriores.

Una vez diseñado los módulos necesarios antes comentados se conectaron usando la herramienta vivado de Xilinx,generando el hardware del sistema para su prueba

La primer prueba que se realizó fue con una señal sin fallas cuyas gráficas se muestran en las figuras 12.

La segunda prueba se realizó generando una falla por hueco de tensión, la cual se puede observar en la figura 13

Las demás fallas se muestran en las figuras 14, 15, 16

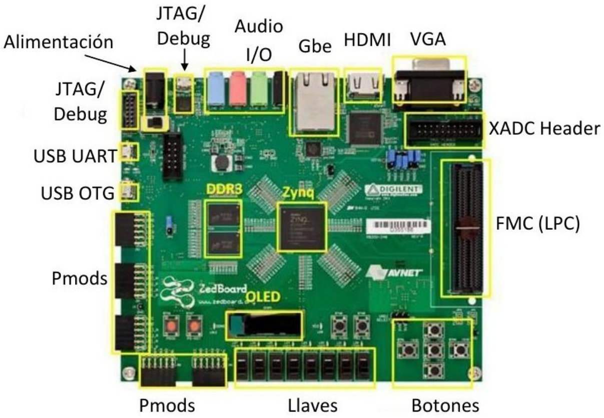


Fig. 11. ZedBoard Zynq Evaluation Developement Kit de Digilent

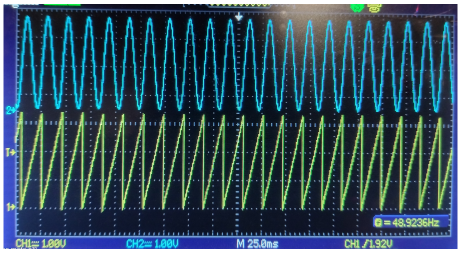


Fig. 12. Señal de red sin fallas.

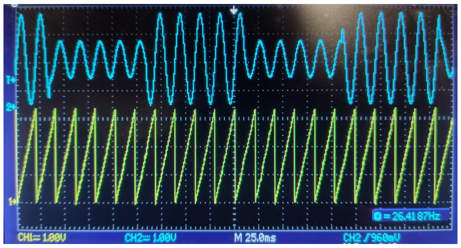


Fig. 13. Señal de red con falla por hueco de tensión

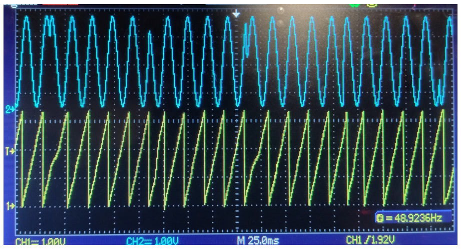


Fig. 14. Señal de red con falla por salto de fase.

References

[1] Daniel Serrano Dominguez. Análisis comparativo de téctnicas de sincronización con la red eléctrica. Trabajo Final de Carrera. Universidad de Sevilla. Julio 2014.

[2] Cynthia Manosalvas Pillajo. Diseño e Implementación de un lazo de enganche de fase (PLL) en un microcontrolador. Trabajo Final de Carrera. Universidad de Zaragoza. 2019.

[3] Algoritmos de detección de fase para sincronización y control de frecuencia de central micro hidráulica plug & play. Trabajo final de carrera. Carlos Patricio Aedo Paredes. Universidad de Chile. Julio 2014

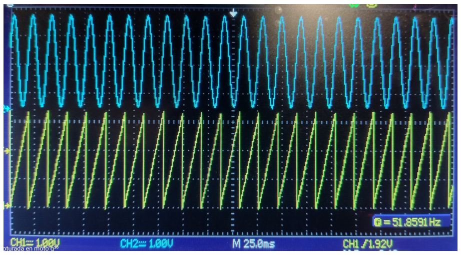


Fig. 15. Señal de red con falla por cambio de frecuencia.

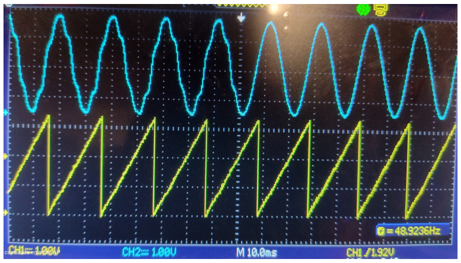


Fig. 16. Señal de red con falla por componente armónico.

[4] Andrés Antonio Segovia Vásquez. Implementación y análisis de algoritmos de sincronización de fase, para fuentes de energía renovables en sistemas trifásicos. Trabajo de titulación previo a la obtención del título de Ingeniero en Electrónica y Telecomunicaciones. Universidad de Cuenca. Julio 2021.

[5] César Augusto Prieto Suárez; Juan David Peña Alvarado. Diseño y Simulación de las Diferentes Etapas de Control en una Micro red Eléctrica. Proyecto Curricular de Ingeniería Electrónica. Universidad Distrital Francisco José de Caldas. 2018.

[6] Alejandro Méndez Samper. Implementación y simulación del algoritmo de sincronización p-PLL mediante un DSP LF2407A. Trabajo de Diploma para optar por el título de Ingeniero Electricista. Universidad Tecnológica de La Habana. Junio 2016.

[7] Antonella Nagliero, Rosa Mastromauro, Marco Liserre, Antonio Dell'Aquila. (2010). Monitoring and synchronization techniques for single-phase PV systems. 1404 - 1409. 10.1109/SPEEDAM.2010.5545057.

Acceleration of a Dense monocular Localization System using FPGAs

***Abstract*—In this work, we accelerate a dense monocular pose estimation system by leveraging the computation capabilities of FPGAs. In a dense monocular setting, pose estimation is usually defined as an minimization problem, where the function to minimize is defined as the normed difference between the observed image and a corresponding image constructed using the estimated pose and a model of the scene. This minimization problem is solved by using a gradient descent method, for example the levenberg-marquardt algorithm. In a dense setting, this requires to compute for every level of detail, for every iteration and for every pixel, the gradient and the hessian of the model with respect to the unknown pose. Also, high frame rates are required in settings where fast and complex camera movements are present, for example for UAV navigation systems. As as consequence, this problem is computationally complex. We accelerate the pose estimation by using the vitis high level synthesis tool, which allows to describe the FPGAs functionality by using C/C++ programming languages. We apply several optimizations and implement the system using a Zynq Ultrascale MPSoC. We compare our results to a pure CPU solution.**

***Index Terms*—Pose estimation, Visual Odometry, FPGAs**

I. Introduction

Currently, robotic systems are in the process of becoming ubiquitous in everyday life. Self driving cars and autonomous vacuum cleaners have commercial products with good performance and are available to middle income costumers worldwide. This shows that there is a clear tendency to continue to use robotics in evermore challenging problems and situations, which require that the robot is in possession, or can acquire, a deep understanding of its surrounding state, and be capable to act upon this information.

The knowledge of a robot pose in relation to the environment in which it must operate is one of the most crucial pieces of knowledge that it must have. Mathematically, this pose belongs to the $SE(3)$ Lie group. It has six degrees of freedom, three of which are related to its position and three to its attitude. In some settings, knowledge of several of these posses must be known to fully describe the robots state. For example, in a robotic manipulator, each of its links has a distinct $SE(3)$ pose. But, nevertheless, the problem can still be described as the estimation of several $SE(3)$ poses. Thus the pose estimation problem in robotics is particularly important.

When the main sensor is a monocular camera, there are already some well established algorithms to recover the poses from several frames captured at different positions [1]. These algorithms have a first stage of image pre-processing to recover a sparse representation of the scene using its most salient features. Common feature extraction and description algorithms usually used in state-of-the-art systems [2] are SIFT [3] and SURF [4]. After extraction, each feature has a R^2 position in the image, that depends on the unknown pose of the observing camera, and a corresponding R^3 position in a common-to-all-cameras reference system.

In a second stage, every R^2 feature observed in a frame is matched with the same R^2 feature observed from another camera pose. Then a minimization problem is constructed, usually called bundle adjustment, with a cost function defined as the normed difference between the R^2 pixel position of the features observed in each frame, and the R^2 feature position given by the model of the scene, namely by the unknown camera poses and the R^3 position of each of the features.

The main drawback of bundle adjustment methods is that they require the 2 pre-processing stages of feature extraction and feature matching. These are normally highly specialized algorithms, in order to perform well specific scene conditions. In this regard, several attempts have arisen in the literature to replace these handcrafted feature extraction and matching algorithms with ones automatically learned by techniques such as deep learning [5].

Other problem is the space nature of the algorithm. The whole of the information present in the image is not used, but a subset represented by the detected features. This information can be sometimes poor, such as in low-gradient images, where very low quantity of features are detected, and in turn making the bundle-adjustment pose estimation difficult.

Other pose estimation systems [6] [7] avoid using these feature extraction pre-processing stages, and instead use directly the values of the pixels observed by the camera.

Again, with the raw pixel observations a minimization problem is crafted, this time the cost function is composed of the normed difference between the raw pixel measurements and the pixel estimation made from the unknown poses of the cameras and a model of the scene. This way, these methods can naturally use every observation made by every pixel to estimate the pose.

However, dense methods require more computing power. This is due to the fact that the gradient and the hessian must be computed for every observed pixel, instead that for every feature extracted from the image.

To mitigate this problem both [6] and [7] use a subset of all the image pixels, by using a strategy to select pixel to be used for the pose estimation. But clearly this undermine the potential robustness gained by these systems that are capable to use every pixel in the image, even those coming from almost flat shaded section of the scene, with contain very low gradient and so very low, but potentially important, information.

In this work, we propose to utilize FPGAs to accelerate a dense monocular pose estimation system. Using High Level Synthesis, we describe the processing system using C and C++ programming languages. We implement several optimization to the baseline code specifically aimed for the FPGA. The proposed pose estimation system is aimed to process $640x480$ images in under 60ms, to achieve 15fps.

II. Mathematical background

The cost function takes the following form:

$$\begin{aligned} C(P_f) &= \sum_{u \in s} \|e(u)\|_h \\ e(u) &= I_f(u_p(P_f, u)) - I_{kf}(u) \end{aligned} \tag{1}$$

Where $P_f \in SE(3)$ is the unknown pose of the f frame which we want to estimate, $I_f \in R^{w \cdot h}$ is the corresponding image observed by the f frame of width w and height h, $I_{kf} \in R^{w \cdot h}$ if a reference frame, for which we known its pose P_{kf} (or we define its pose as the identity pose) and have for each of its pixels $u = (u_1, u_2) \in R^2$ the inverse depth id, the distance in the z direction between the focal point and the corresponding R^3 point for a particular pixel u. Finally, $u_p(P_f, u)$ is the corresponding pixel in the frame f which depends on the particular pixel u in kf and the pose P_f, and $\|\|_h$ is the huber norm.

The coordinates of the pixel u_p in the f frame, corresponding to the pixel u as seen from the frame of reference kf can be calculated using

$$\begin{aligned} u_p(P_f, u) &= \pi(Kp_f) \\ p_f &= P_f P_{kf} K^{-1} \pi^{-1}(u)/id(u) \end{aligned} \tag{2}$$

where $\pi(x, y, z) = (x/z, y/z)$, $\pi^{-1}(u_1, u_2) = (u_1, u_2, 1)$, and

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} (3) \quad K^{-1} = \begin{bmatrix} if_x & 0 & ic_x \\ 0 & if_y & ic_y \\ 0 & 0 & 1 \end{bmatrix} (4)$$

is the camera intrinsic matrix and its inverse, respectively. To estimate P_f we use the Levenberg-Mardquat algorithm to solve (1). The jacobian $J^k = \frac{\partial I_f}{\partial P_f^k}$, for the kth iteration is computed using

$$\begin{aligned} J^k(u) = (P_0^k(u), P_1^k(u), P_2^k(u), &P_3^k(u), P_4^k(u), P_5^k(u)) \\ P_0^k(u) &= \frac{\partial I_f(u)}{\partial u_1} f_x / p_{fz}^k(u) \\ P_1^k(u) &= \frac{\partial I_f(u)}{\partial u_2} f_y / p_{fz}^k(u) \\ P_2^k &= -(P_0 p_{fx}^k(u) + v1 p_{fy}^k(u)) / p_{fz}^k(u) \\ P_3^k &= -p_{fz}^k(u) \cdot P_1 + p_{fy}^k(u) P_2 \\ P_4^k(u) &= p_{fz}^k(u) P_0 - p_{fx}^k(u) P_2 \\ P_5^k(u) &= -p_{fy}^k(u) P_0 + p_{fx}^k(u) P_1 \end{aligned} \tag{5}$$

where $p_f^k(u) = (p_{fx}^k(u), p_{fy}^k(u), p_{fz}^k(u))$ is the point $\int R^3$ corresponding to a particular pixel u and its inverse depth $id(u)$ as seen in the frame f as defined in (2).

Finally, we compute

$$\begin{aligned} g^k &= \sum_{u \in s} J^k(u) \cdot e(u) \\ H^k &= \sum_{u \in s} J^k(u) \cdot J(u)^{kT} \end{aligned} \tag{6}$$

where g^k is the gradient vector $\in R^6$ and $H^k \in R^{6x6}$ is the Hessian matrix, both for iteration k. The update δP_f^k is computed solving a system of linear equations

$$H^k \delta P_f^k = g^k \tag{7}$$

and finally the estimate for iteration $k{+}1$, P_f^{k+1} is computed as

$$P_f^{k+1} = P_f^k + \delta P_f \tag{8}$$

this is done iteratively until some convergence condition is reached. In our case, we check

- if the norm of δP_f is lower that $1e - 16$
- if the $C(P_f^{k+1})/C(P_f^k)$, as defined in 1, is lower than 0.999

The algorithm starts with an initial guess of $P_f^0 = P_{kf}$. To improve its robustness, a level of detail (LoD) approach is used [6]. Both I_{kf} and I_f are down-sampled in several levels (1/2 resolution, 1/4 resolution and so forth) and K and K^{-1} are scaled accordingly. The system starts with the lower level of detail, and the resulting P_f is feed back as initial guess to higher levels of image resolution, until the level of detail corresponding to the original image resolution is reached.

III. System architecture

The computationally demanding portion of the algorithm is concentrated in the computation of (6), which must be done several times, both for each level of detail and for each iteration within each level.

The literature suggests the use of 5 levels of detail [6], with iterations ranging from 5 for the finest level and 100 for the coarsest one. This means that (6) will have to be computed hundreds of times for each new frame f.

On the other hand, computing (7) involves solving a system of equations, which can be done quite fast using a CPU and some specialized linear algebra libraries such as Eigen [8].

Other sections of the algorithm, such as the acquisition of the images, the check of the convergence condition, data movement from the image acquisition system and the FPGA can be also efficiently commanded by the CPU, using specialized libraries like OpenCV [9] and the DMA already present in the CPU.

As a consequence, the computation will be done in a hybrid fashion using the CPU and the FPGA. This way, the system leverages the strength of the CPU in the sequential portions of the algorithm, and the FPGA in the highly parallel numerical computations required to solve (6).

A step by step description of the algorithm is as follows:

1) The CPU sequentially acquires the input frames, being from a camera or a file system.
2) The data pre-processing stages, such as the downsampling of the images and the movement of data between CPU and FPGA will also be done through the CPU.
3) For each LoD:
4) For each iteration:
5) The CPU instructs the FPGA to compute H and g according to (6), and waits until this computation is done.
6) The CPU solves (7) and updates P_f according to (8), checking if the convergence condition was met.

A CPU implementation for (6) was developed, to function as gold standard to compare both results with the FPGA as well as to compare computing time.

A. CPU implementation

The CPU implementation of (6) was done in a straightforward fashion. All the optimization steps are accomplished using specialised external libraries.

The function iterates through all the pixels u of the reference frame kf, computes $J(u)$ and accumulates g and H. The linear algebra computations, such as vector addition, matrix multiplication and $SE(3)$ operations were implemented using the Eigen library [8], which is highly optimized both for CPUs with ARM based architectures or x86 based CPU architectures.

Floating point arithmetic was used through the whole implementation. It is important to note that data movement is not required beyond the acquisition stage, where the image information is moved from the acquisition hardware to the CPUs RAM. We used just one CPU to iterate through all pixels u. This way, we can estimate a computing time gain of at most n, being n the number of CPUs that the particular system architecture has.

B. FPGA implementation

The FPGA used to developed this system was the `Zynq Ultrascale+ MPsoC`. The FPGA implementation of (6) was done using the *vitis hls* compiler provided by Xilinx, which allows to describe the FPGA configuration through the use of C and C++ languages. It is important to notice that direct use of the CPU C/C++ code is not possible. This is in part because of the use of dynamic memory is not possible in a FPGA setting, part because most of the libraries used in the CPU implementation, such as [8], is not synthetisable, and part because of the intrinsic differences between CPUs and FPGAs requires to develop especially tailored code to have good performance on either system.

1) System inputs and outputs: To compute (6), the FPGA requires I_{kf}, which is an 8bit $h \cdot w$ matrix, I_f which is also a 8bit $h \cdot w$ matrix, and id_{kf}, which is a floating point image containing the inverse depths for each pixel in kf, so it is a 32 bit $h \cdot w$ matrix. Also, the current estimated pose P_f^k is needed, and the intrinsic camera matrix K and K^{-1}. All these data must be passed from the CPU RAM to the FPGA. The size of h and w will change depending of the LoD being use at the moment.

As result, the FPGA will compute g and H, and this data must be given back to the CPU.

2) Naive implementation: As first step toward an FPGA implementation, we took the CPU code and made the essential modifications to develop a synthetisable code.

To support linear algebra operations such as the $SE(3)$ operations in the FPGA, we programmed from the ground up our own library specially tailored for FPGA syntheses. The $SE(3)$ Lie group was represented internally as a rotation matrix $\in R^{3x3}$ and a translation vector $\in R^3$. Vectors and matrices $\in R^2$, $\in R^3$ and $\in R^{3x3}$ are all defined as well as its interoperability with the $SE(3)$ group. Also we programmed operations $\in R^6$ and $\in R^{6x6}$ to be used for g and H.

The system computes g and H mainly through the following steps:

1) Define H and g, initialize with 0
2) Iterate through the y image dimension, from 0 to h
3) Iterate through the x image dimension, from 0 to w
4) Compute H_u and g_u for the pixel (x,y)
5) Accumulate H_u and g_u in H and g

The system input-output was accomplished through direct RAM access. This way, in stage 4, the FPGA read from RAM $I_{kf}(u)$, $id(u)$ and $I_f(u_p)$.

To measure the implementation performance, the Vitis compiler needs to know at compile time the number of iterations for each loop. As h and w are variables, that depend on the specific LoD that is being computed at the time, we use the pre-procesor directive `#pragma HLS loop_tripcount` to tell the compiler the values that h and w can take. We set $min = 480$ and $max = 480$ for the y iteration and $min = 640$ and $max = 640$ for the x iteration. This way, the performance will be measured for a image of 640x480, which is the native resolution of our test dataset. This also means that all performance metrics will be measured for the first LoD.

This implementation has a latency of 12288464 clock cycles, that for the $150MHz$ clock that the used FPGA has, translate to a latency time of 81.27 ms. The Vitis compiler automatically flattened and pipelined the y-x loop, which completes its computation in 12288262 cycles. Besides some cycles used in initialization stages, the y-x loop takes the majority of the computation. The pipeline initialization interval is of 40 cycles, which means that there is data-dependencies in the y-x loop that does not allow the pipeline to run one computation per clock.

C. Optimizations

The first optimization was aimed at reducing the data dependencies in the y-x loop. The reports given by the Vitis compiler shows that the accumulation in step 5 is the limiting factor in improving the pipeline initialization time. Indeed, if the accumulation of H or g is not finished, the next pipeline stage cannot begin. As H is a R^{6x6} matrix, its accumulation involves 36 loads, 36 adds and 36 stores. On the other hand, g is a R^6 vector and so it involves 6 loads, 6 adds and 6 stores. These adds can be done all in parallel, but the Vitis tool indicates that the H data structure was implemented using BRAM. This limits the parallel loads and stores possible to just 2.

A first solution was to use the pre compiler instruction `#pragma HLS array_partition complete`. This tells the Vitis compiler to synthetise the whole array of data as internal logic, instead of using BRAM, which would allow to have more loads and stores per cycle. This first solution did not give good results, because H and g are represented using floating points. The floating point add function requires 12 clock cycles, so the data dependency remains because even when the load and store can be done in 1 clock cycle, the add operation cannot be completed. To solve this issue, instead of having just one H and g accumulator, we implemented the system with an array of 16 accumulation $H[16]$ and $g[16]$. In each clock cycle the array index is changed. This solves the data dependency and allows the pipeline to run at 1 cycle initialization interval. After we finish computing the y-x loop, we run an additional step of adding the 16 H and g together.

This implementation has a clock latency of 1537026 cycles, which translates to a time latency of 10.247 ms. The y-x mloop was again flatten and pipelined by the Vitis compiler, taking 1536262 clock cycles to compute, again mainly all of the clock cycles are used to compute the y-x loop. The pipeline initialization interval is of 5 cycles.

This time, the data dependency in the pipeline of the y-x loop is in the reading the I_f. Computing H and g as described by 5 requires to access to the u_p value of I_f, as well as to access the derivatives $\frac{\partial I_f}{\partial u_p}$. These derivatives are approximated by computing $\frac{\partial I_f}{\partial u_1} = (I_f(u_1+1, u_2) - I_f(u_1-1, u_2))/2$ and $\frac{\partial I_f}{\partial u_1} = (I_f(u_1, u_2+1) - I_f(u_1, u_2-1))/2$. This requires to access 5 different values of I_f, namely $I_f(u_1, u_2)$, $I_f(u_1+1, u_2)$, $I_f(u_1-1, u_2)$, $I_f(u_1, u_2+1)$, $I_f(u_1, u_2-1)$. These values are hard to be read from RAM using a burst-read operation, because $u_p = (u_1, u_2)$ is not known beforehand, it is computed using 2.

To solve this, before the y-x loop we read the whole I_f to BRAM. This operation can be done in a RAM burst-read operation, and having the whole I_f allows to read in an almost random access manner the pixels locations u_p computed by 5.

In addition, constantly reading from RAM the values from I_{kf} and id_{kf} is not optimal. It would be better to read in a burst-operation several values for both of these data structures from RAM and store them in BRAM, and compute the rest of the pipeline using these cached values. We implemented 2 internal BRAM memories, to store the y and $y+1$ line of both I_{kf} and id_{kf}. Before the stage 2 we read the $0th$ line, and in stage 4 we compute the H and g values for the yth line and the xth column using the BRAM memory for the yth line, and simultaneously read the y+1 line from RAM to BRAM.

The resulting implementation is described using the following stages

1) Define a BRAM to store two lines of I_{kf} and id_{kf}, and the whole I_f image
2) Define $H[16]$ and $g[16]$ data structures, initialized to 0
3) Read from RAM the $0th$ row of size w of I_{kf} and id_{kf}, and store it in BRAM
4) Read the whole I_f and store it in BRAM
5) Iterate through the y image dimension, from 0 to h
6) Iterate through the x image dimension, from 0 to w
7) Compute Hn and gn for the pixel (x,y) using yth row of BRAM-stored I_{kf} and id_{kf}
8) Store in BRAM the x value of the (y+1)th row of I_{kf} and id_{kf}
9) Save Hn and gn in $H(x\%16)$ and $g(x\%16)$
10) Add together the 16 values of H and g.

This implementation has a clock latency of 959655, and a time latency of 6.398 ms. The reading of I_f to BRAM takes 343681 cycles, that for a 640x480 image is almost 1 clock per pixel. The y-x loop takes 614532 cycles, with an pipeline initialization interval of 2 cycles. This 2 cycles result from the 5 reads that must be performed for the I_f. We use the `#pragma HLS array_partition cycle` pre compiler directive for I_f BRAM data structure, to store the different y lines of I_f in different BRAMS, but still we could not avoid the need to read 3 simulaneous pixel values, that can be done at a minimum in 2 clock cycles.

From these results, we can compute 1 iteration in the first LoD (640x480) in 6ms, the second LoD level (320x240) can be computed in 3ms, the third one in 0.75ms, the forth in 0.18ms and the fifth in 0.04 ms. If we take the maximum iterations allowed for each levels used in [6], namely 0, 10, 20, 50, 100, the system will estimate a pose in 58ms, achieving 15 fps.

The FPGA resource usage is summarized in the table III-C.

	BRAM	DSP	FF	LUT
Used	395	169	49857	41562
Available	624	1728	460800	230400
Percentage	63	9	10	18

As can be seen, the usage of the FPGA is well withing the maximum resources available. The resource most utilized is clearly the BRAM, which is natural from the highly memory bound nature of the algorithm.

IV. Results

In this section, we present the results of our experiments testing the performance of the FPGA-based dense monocular pose estimation system we developed. We used a synthetic dataset for testing, where we had access to the true pose for each frame. The dataset contained an office scene, including desks, PCs, chairs, and bookshelves. An example image from the dataset is shown in Figure 1.

To evaluate the performance of the system, we computed the id using the SLAM system DTAM [10] for some of the images in the dataset. These images were used as reference frames during the execution of the system. We saved both the frames in the dataset and the inverse depths of the reference

Fig. 1: Example image of the dataset used to test the system.

frames in the SD card, which contained the Linux filesystem for the FPGA, as well as the FPGA kernel code and the CPU program.

We compared the performance of our FPGA-based system with that of a single-core CPU implementation. The mean time taken by the CPU pose estimation to process each frame was 103.4 ms, while the FPGA implementation took a mean of 75 ms to compute. The additional time taken by the FPGA implementation can be accounted for by the extra time taken by the CPU to accomplish the RAM-FPGA memory communication.

Our experiments showed that the FPGA-based system we developed was able to outperform the single-core CPU implementation, achieving 13 fps. While these results are promising, there is still room for further optimization. In particular, we believe that wider RAM to FPGA memory accesses and a better BRAM access pattern could further improve the system's performance.

V. Conclusions

In this work, we developed a FPGA-based dense monocular pose estimation system that outperformed a single-core CPU implementation. Our system achieved a frame rate of 13 fps and showed promising results in the synthetic dataset we used for testing.

Our results demonstrate the potential of using FPGA-based systems for dense monocular pose estimation, which has important applications in robotics, augmented reality, and virtual reality. In particular, FPGA-based systems could enable real-time performance in these applications, which is critical for user experience and safety.

While our system achieved good performance, there is still room for improvement. In particular, future work could focus on optimizing the system's memory access patterns and exploring alternative algorithms for dense monocular pose estimation. We believe that these improvements could further enhance the performance of the system and make it even more useful for real-world applications.

Conflict of interests

Authors declare no conflict of interests.

References

[1] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment - a modern synthesis," in *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, ser. ICCV '99. London, UK, UK: Springer-Verlag, 2000, pp. 298–372. [Online]. Available: http://dl.acm.org/citation.cfm?id=646271.685629

[2] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "Orb-slam: A versatile and accurate monocular slam system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, Oct 2015.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, nov 2004. [Online]. Available: http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94

[4] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Surf: Speeded up robust features," *Computer Vision and Image Understanding (CVIU)*, vol. 110, no. 3, pp. 346–359, 2008.

[5] J. Sun, Z. Shen, Y. Wang, H. Bao, and X. Zhou, "Loftr: Detector-free local feature matching with transformers," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 8922–8931.

[6] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-scale direct monocular SLAM," in *European Conference on Computer Vision (ECCV)*, September 2014.

[7] J. Engel, V. Koltun, and D. Cremers, "Direct sparse odometry," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Mar. 2018.

[8] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[9] Itseez, "Open source computer vision library," https://github.com/itseez/opencv, 2015.

[10] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison, "Dtam: Dense tracking and mapping in real-time," in *2011 International Conference on Computer Vision*, Nov 2011, pp. 2320–2327.

XI Southern Conference on Programmable Logic SPL2023

Table of Contents Design Forum

FPGA-Accelerated Convolutional Neural Network

1st Mohammed CHELKHA
SEMi, SIGER
FMPs, USMBA
Brussels, Belgium
mohammed.chelkha@umons.student.ac.be

2nd Carlos VALDERRAMA
ELECTRONICS AND MICROELECTRONIcS UNIT
POLYTECHNIC FACULTY OF MONS
Mons, Belgium
carlos.valderrama@umons.ac.be

3rd Ali AHAITOUF
SIGER
USMBA
Fez, Morocco
ali.ahaitouf@usmba.ac.ma

***Abstract*—In recent years, FPGA has become an attractive solution to accelerate CNN classification for its flexibility, short time-to-market, and energy efficiency. The real-time evaluation of a CNN for image classification on a live video stream can require billions or trillions of operations per second. To come with a competitive re-configurable implementation satisfying both development time and flexibility, we propose using as a base a re-configurable Architecture composed by a set of image and video processing blocks. The whole architecture can be configured on-the-fly based on the image characteristics thus supporting variable image resolutions for each layer of the CNN.**

***Index Terms*—Convolutional Neural Network, FPGA, Deep learning, Coarse-Grain**

I. Introduction

Convolutional Neural Networks have been some of the most influential innovations in the field of computer vision. [1] 2012 was the first year that neural nets grew to prominence as Alex Krizhevsky used them to win that year's ImageNet competition (the annual Olympics of computer vision), dropping the classification error record from 26 to 15 percent, an astounding improvement at the time.

CNNs extraordinary performance comes at a high cost in terms of computing complexity. Even more effort is required for picture segmentation and scene tagging. While the latest graphics processing units (GPUs) can achieve this level of speed, there is also a need to integrate such solutions into other systems, such as vehicles, drones, or even wearable gadgets, which have stringent physical size and energy consumption constraints. As a result, future embedded CNNs will require compact, efficient, yet extremely powerful processing platforms. [2] This study aims to explore the possibilities of utilizing an existing novel flexible architecture for real time image and video processing, that takes advantage of the inherent parallelism of Filed programmable Gate Arrays (FPGAs) to achieve real-time performance. We can resume our proposal in two main contributions:

- A convolutional neural network architecture that has been designed to fit perfectly on the FPGA. The CNN is very consistent, and it achieves a satisfactory classification accuracy with low processing cost.
- A hardware/software co-design to efficiently accelerate the entire CNN on FPGAs. We propose a uniformed convolutional matrix-multiplication representation for both computation-intensive convolutional layers and pooling layers.

II. Related Work

GPUs were initially developed to accelerate graphics processing. A GPU is particularly designed for integrated transform, lighting, triangle setup/clipping, and rendering. A modern GPU is not only a powerful graphics engine but also a highly parallelized computing processor featuring high throughput and high memory bandwidth for massive parallel algorithms [3], which is dubbed as GPU computing or general-purpose computing on GPU (GPGPU). For our interest, CNNs can take advantages of the nature of algorithmic parallelism in the following aspects [4] : (i) the convolution operation of an n x n matrix using a k x k kernel can be in parallel; (ii) the sub-sampling/pooling operation can be parallelized by executing different pooling operations separately; (iii) the activation of each neuron in a fully connected layer can be parallelized by creating a binary-tree multiplier. With great parallel processing structures and strong floating-point capabilities, GPGPUs have been recognized to be a good fit to accelerate deep learning. A number of GPU-based CNN libraries have been developed to facilitate highly optimized CNN implementation on GPUs, including cuDNN [5], Cuda-convnet [6] and several other libraries built upon the popular deep learning frameworks, such as Caffe [7], Torch, Tensorflow [8], Keras, etc.

III. Background

As a typical supervised learning algorithm, there are two major phases in CNN: training phase and inference (aka feed-forward) phase. Since many industry applications would train CNN in the background and only perform inferences in a real-time scenario, we mainly focus on the inference phase in this thesis. The aim of the CNN inference phase is to get a correct inference of classification for input images. It is composed of multiple layers, where each image is fed to the first layer. Each layer receives a number of feature maps from a previous layer, and outputs a new set of feature maps after filtering by certain kernels. The convolutional layer, activation layer, and pooling layer are for feature map extraction, and the fully connected layers are for classification. When these layers are stacked, we have formed a full CNN architecture. However, knowing the overview of how CNNs operate isnt going to be sufficient to implement a CNN with real world data. Its imperative to not only understand the individual layers, but the fine points of the parameters and how they communicate with other layers too.

IV. Enhanced P2IP Architecture for Real-World Application

The P2IP is a systolic CGRA designed for real-time image and video processing targeting embedded systems. The objective of this architecture is to overcome the limitations of the existent solutions for image and video processing, combining high-performance, low-latency, and low-power consumption, with a level of flexibility. Images or frames are entered as a stream of pixels in a sequential line-scanned format progressing through the pipeline at a constant rate. The P2IP datapath works at the pixel clock frequency,delivering one processed pixel per clock cycle after a initial latency to fill the pipeline.It was projected to work between a frame source and a frame sink directly on the pixelstream. In order to allow the P2IP integration into an image processing chain, the AXI4-Stream [12] was adopted as the external interconnection protocol. The AXI4-Stream protocol is managed by the P2IP Controller which is also in charge of reading the configuration words on the configuration input port (config in) and transferring them to the processing core. The P2IP controller is the input of the P2IP configuration mechanism followed by a Configuration Decoder Tree (CDT), distributed throughout the processing core.

V. Hardware Implementation of P2IP-CNN on FPGA

A. Developping the Baseline Model

The design of the test harness is modular, and we can develop a separate function for each piece. This allows a given aspect of the test harness to be modified or inter-changed, if we desire, separately from the rest.

We can develop this test harness with three key elements. They are the preparation of the dataset, the definition of the model and the extraction of the weights and the feature maps of each layer.

B. Accelerator Design Exploration

Our CNN accelerator design on FPGA is composed of several major components, which are processing elements (PEs), on-chip buffer, external memory, and on-/off-chip interconnect. A PE is the basic computation unit for convolution and pooling. All data for processing are stored in external memory. Due to on-chip resource limitation, data are first cached in on-chip buffers before being fed to PEs. Double buffers are used to cover computation time with data transfer time. The on-chip interconnect is dedicated for data communication between PEs and on-chip buffer banks.

There are several design challenges that obstacle an efficient CNN accelerator design on an FPGA platform. First, loop tiling is mandatory to fit a small portion of data on-chip. An improper tiling may degrade the efficiency of data reuse and parallelism of data processing. Second, the organization of PEs and buffer banks and interconnects between them should be carefully considered in order to process on-chip data efficiently. Third, the data processing throughput of PEs should match the off-chip bandwidth provided by the FPGA platform. The two-level unrolled loops are implemented as concurrently executing computation engines and a tree-shaped poly structure is used. For the best cross-layer design case, the computation engine is implemented as a tree-shaped poly structure with 9 inputs from input feature maps and 9 inputs from weights and one input from bias, which is stored in the buffers of output feature maps. This architecture consists of two parts: a data access system and a PE array. The data access system includes two parts, namely, a DDR3 controller and a Cache IP. The DDR3 controller is used to exchange data between the DDR3 memory and the Cache IP. The Cache IP can provide a feature map, kernel, and weight to the function module. The PE array is the computation core of the accelerator and consists of function modules and reorganization buffers. Each PE is a pipeline architecture, the execution time between two adjacent PEs is the super-pipeline cycle. The controller is not a part of the accelerator and is used to interact with the CPU and accelerator.

C. Design of the HW-SW platform

The rapid evolution of system-on-a-chip technologies has created the need for hardware-software co-design since these two constituent elements (HW-SW) of modern embedded systems can no longer be treated separately. This forms an important gap in existing methodologies, since the designer would like to be able to evaluate a number of alternative architectures, before committing to a specific one.The design of an embedded system using a co-design approach, involves a series of actions that must be followed [13] In hardware-software co-design approaches, the whole development cycle should be based around a single model.This model evolves during the various design stages from the initial informal conceptualization of the user's requirements to the final implementation-level detailed description of the system. The next step is to refine the formal system specification so that all details - including implementation decisions - are contained in the system model. Finally, the emerged system model is translated to implementation languages like C, C++, Java etc. for software and VHDL, Verilog, Hardware C etc. for custom hardware. Our system does this by using a hybrid of layer and model parallelism together with a number of new workload/weight balancing strategies. a single on-the-fly reconfiguration is needed: each configuration computes certain layers, or a part of a single layer; each device is optimized independently with respect to its own computation.

We find this approach to be effective with performance similar to that of GPU clusters of similar size and technology,but with far better power efficiency. The limiting factor is inter-FPGA bandwidth. The framework for mapping CNN logic to distributed FPGA clusters that achieves both high efficiency and scalability; that does not suffer from issues related to mini-batch size; and that needs only a simple interconnection network as is available in any multi-FPGA system with consistent communication and reasonable bandwidth. To ensure the connection, we will use the PIO (Parallel I/O) component from Platform Designer. When adding the component, we get to choose the direction and the width of the register, also the base address of PIO component which is very important. The ARM program will access the component according to this base address. The ARM program development will make use of this address and a given Linux shell batch file will help extract the address information to a header file hps_0.h.

Finally, we need to integrate the SoC design with our P2IP using Verilog code to instantiate the core; The Verilog code generated and modified still has to be compiled into a bit-stream for the FPGA. With the schedule and resource allocation already fixed and all timing constraints properly set, there is not much left to be configured in Quartus itself. However, the timing results reported by the Quartus can be quite different from the estimates reported by the simulation in ModelSim.

D. ARM program development

With all these steps done, the FPGA side of the CNN accelerator is complete. However, there is still a missing key component: The CPU-side software which will run the rest of the CNN layers and configure the P2IP at each step . This subsection introduces how to design an ARM C program to control the CNN FPGA-Accelerator. Altera SoC EDS is used to compile the C project. For ARM program to control the P2IP component, the registers addresses are required. The Linux built-in driver '/dev/mem' and mmap system call are used to map the physical base address of P2IP component to a virtual address which can be directly accessed by Linux application software.

VI. RESULTS

A. P2IP CNN FPGA Accelerator Performance

The P2IP CNN FPGA Accelerator is meant to be a proof-of-concept for the implementation of CNNs on the basis of a systolic CGRA for image and video processing on FPGA. The secondary goal targets a maximum throughput on the given small and low-power platform, and in consequence a good power efficiency. This section evaluates the finished design with regard to the factors resource utilization, accuracy and the throughput of the accelerator. Finally, a number of potential architectural optimizations are highlighted.

The P2IP Embedded CNN has been completely assembled and successfully taken into operation on a DE1 SoC Board. The full test system consists of :

- HPS (ARM Cortex9 with 1 GB DDR3 memory), running under Linux4.
- MNIST CNN network description and trained weights, copied to the memory
- P2IP CNN FPGA Accelerator bitstream, loaded into the FPGA fabric
- P2IP CPU-side application, feeding the input images, launching the FPGA accelerator, measuring the timing and checking the classification results.

Using the above system configuration, the P2IP Embedded CNN has been evaluated in a realistic embedded scenario.

B. Throughput and Latency

The embedded CNN's throughput is measured in terms of images per second. In a typical scenario, the CNN accelerator is configured with the network description and the trained weights beforehand, and is then utilized to classify an incoming stream of images. Therefore,the run-time per frame is measured from the moment when the FPGA accelerator is started,to the moment when the calculation of the Softmax Classification layer is finished. In the P2IP, each PE can have a different latency according to its configuration.

The NE latency (NEL) for a neighborhood window with m × n pixels in an image with dimensions M × N pixels, can be expressed as defined here :

$$NEL = N(\frac{m-1}{2}) + \frac{n+1}{2} + b$$

where N is the number of pixels in a image line, m is the number of lines in a neighborhood window, n is the number of pixels per neighborhood line, and b is the border handler latency. The Convolver Module latency for a 3x3 filter is calculated as 9 pixel clock cycles and the Reconfigurable Interconnection latency as 6 pixel clock cycles.

C. Accuracy and Error

the Keras Python CNN model scored a 94% accuracy in the test. To put the P2IP CNN model to test and confirm these numbers, we extracted the feature maps of each layer and did a similarity comparison with the Keras model. This was achived using the SSIM algorithm.

The SSIM was first introduced in the 2004 IEEE paper [14] , it was introduced as an alternative complementary framework for quality assessment based on the degradation of structural information. The Structural Similarity Index (SSIM) metric extracts 3 key features from an image: Luminance, Contrast, Structure. The comparison between the two images is performed on the basis of these 3 features. This was all implemented in Python and the scores were ranging from 0.81 to 0.68 in terms of similarity.

D. Resources Analysis

Regarding the amount of logic required by the P2IP architecture, the Table I presents the resources required by the proposed implementations. Note that since all devices from the "V" series share the same internal architecture,the resources utilization is similar for both boards.

VII. DISCUSSIONS

A. Comparison with State-of-the-Art Architectures

we confine ourselves to a summary of the most important characteristics in this section. To start with, table II repeats the comparison of the different CNN topologies, and this time includes the P2IP CNN and its key parameters. All the parameters from our CNN, were calculated based on the Pyhton-Keras model.

The comparison presented in this section is concentrated on programmable architectures that target embedded systems, performing image classification.

Components	Adaptive Logic Module	Memory (kbits)	DSP
Control and Interface			
Controller	151	0	0
Input Register	15	up to 256 MB	0
Output Register	12	up tp 256 MB	0
Total	178	-	0
PE			
PE-CD	38	0	0
Spatial processor	1221	2.3	14
Memory Controller	1838	131.1	0
Reconfigurable Interconnection	211	0	0

TABLE I
P2IP CNN RESOURCES REQUIREMENTS ON FPGA DEVICES FROM THE ALTERA CYCLONE"V" SERIES

	Conv Layers	MACCs (millions)	params (millions)	Error
P2IP CNN	Up to 20	530	1.1	19%
AlexNet	5	1140	62.4	19.7%
VGG-16	16	15470	138.3	8.1%
GoogleLeNet	22	1600	7.0	9.2%
ResNet-50	50	3870	25.6	7.0%
SqueezeNet	18	860	1.2	19.7%

TABLE II
COMPARISON OF P2IP CNN TO CNN ARCHITECTURES FROM PRIOR WORK.

the P2IP CNN has an evident advantage over the CPU implementation. The P2IP speedup factor for the CPU implementation varies from 24 to 29. Regarding the GPU implementation, the P2IP CNN presents an advantage on almost all resolutions. The GPU tends to have an increasingly better performance in function of the image resolution, while the P2IP has a constant performance.

In addition to its competitive performance, the P2IP can still offer portability and much lower power consumption when compared to GPUs and CPUs. The P2IP solution consumes 50 times less power than the CPU implementation and 90 times less power than the GPU implementation.

B. Potential Improvements

An architectural bottleneck can be seen in the prefetching of image pixels from the image cache. Although this task is executed in parallel to the actual output channel calculation to hide the prefetch latency, the current delay is relatively long. The architecture of the image cache might need to be improved to allow for more parallel read accesses, or a register-field might be used to cache the active image patch. This should be viable as the image cache occupies less than 8 % of the Block ROMs, and a total of 300 k flip-flops are still unused. An ideal image cache would have a latency of less than 5 clock cycles, which would result in a speedup factor of 1.4.

Also, 5×5 convolutions are currently not implemented efficiently: two 3×3 MACC units are used for the necessary multiplications. The potential overall speedup from utilizing all 18 multipliers in the MACC units for individual 5×5 convolutions is approximately 1.2 with the current prefetch latency of 9 clock cycles. With an ideal Image Cache, a speedup factor of nearly 1.5 could be achieved.

VIII. CONCLUSION

In this article, we demonstrated the design and implementation of a proof-of-concept FPGA-accelerated embedded Convolutional Neural Network. The P2IP CNN is designed for image classification and consists of two main components: a highly optimized and customized CNN FPGA topology, and the P2IP CNN HW-SW platform, a FPGA-based architecture for configuring and running the FPGA accelerator. The P2IP Embedded CNN has been assembled into a fully working proof-of-concept system on both the DE-1 and the SoCkit board, All Programmable platforms. This project clearly demonstrates the feasibility of FPGA-based embedded CNN implementations. The current solution already exhibits a reasonable performance, and a number of opportunities for further gains in throughput and power efficiency have been pointed out.

The tough requirements of embedded CNNs regarding the size, efficiency and computational power of the underlying computing platform are very hard to meet with the systems available today. Even though the presented P2IP CNN does not yet provide the massive amounts of computational power required for future applications of embedded image understanding, it may still serve as a steppingstone and a guide for further explorations of the FPGA as a platform for embedded CNNs. The biggest advantage of these FPGA-based systems can be seen in their scalability. Using a larger device, much higher performance can be attained at comparable efficiency figures,while most other platforms are inherently limited by the amount of computational power available on a given chip. FPGAs therefore provide a promising path towards the vision of powerful embedded CNNs and the abundance of fascinating applications which could profit from on-board image understanding

REFERENCES

[1] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, vol. 5, no. 4, pp. 115–133,, 1943.

[2] A. Karpathy. Surpassing human-level performance on imagenet classification, 2014.

[3] WikiPedia. Graphics processing unit, 2013.

[4] Magnus Halvorsen. Hardware acceleration of convolutional neural networks. MS thesis, Norwegian University of Science Technology, 2015.

[5] haran Chetlur. Cudnn: Efficient primitives for deep learning. arXiv: 1410.0759, 2014.

[6] Alex Krizhevsky. Cudaconvet2, 2013.

[7] Yangqing Jia. Caffe: convolutional architecture for fast feature embedding. International Conference on Multimedia, 2014.

[8] Tensorflow, https://www.tensorflow.org/ 2014.

[9] cs231n. Convolutional neural networks for visual recognition, 2016.

[10] D. Gschwend; C. Mayer; S. Willi. Design and implementation of a convolutional neural network accelerator asic. Semester Thesis, ETH Zürich, 2015.

[11] Keiron O'Shea; Ryan Nash. An introduction to convolutional neural networks, 2015.

[12] ARM , AXI4 , https://www.arm.com, 2010

[13] F. Vahid D. Gajski. Specification and design of em-bedded hardware-software systems. IEEE Designand Test of Computers, vol.12. pp.53-67, 1995.

[14] Zhou Wang . Alan Conrad Bovik . Hamid Rahim Sheikh. Image quality assessment: From error visibility to structural similarity. IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 13, NO. 4, 2004.

Prototipo para estudio del timing de un convertidor analógico-digital de aproximaciones sucesivas

Daniel Crepaldo, Carlos Varela, Lisandro Martín,
Federico Pacher, Eduardo Bailón, Javier Ghorghor.
Laboratorio de Microelectrónica - FCEIA - Universidad Nacional de Rosario
Rosario, Argentina
microlab@fceia.unr.edu.ar

***Abstract*— Se presenta un prototipo desarrollado en tecnología mixta para el estudio y optimización funcional y de temporización de un convertidor analógico-digital de aproximaciones sucesivas por reparto de carga como paso previo a su diseño e implementación como parte de un ASIC en tecnología CMOS. El circuito de control del convertidor se implementó utilizando una placa de desarrollo que incluye una FPGA Spartan 3E, mientras que el bloque analógico se realizó con elementos discretos montados en una plaqueta para evaluar en campo el impacto de los distintos parámetros de funcionamiento (valor de los capacitores y resistencia de las llaves, frecuencia, retardos, solapamiento de llaves, consumo, etc.) en el funcionamiento general del CAD. El prototipo permitirá verificar la influencia de los valores de los capacitores, la impedancia de las llaves analógicas y los tiempos de respuesta del resto de los componentes del circuito en la redistribución de la carga del convertidor y sus prestaciones finales.**

Keywords*— Convertidor A/D, timing, mínimo consumo, tasa conversión, tecnología CMOS.

I. Introducción

Los sensores incluidos en los nodos de una red inalámbrica inteligente reconfigurable encargada de monitorear y recolectar datos de variables climáticas (temperatura, humedad, presión, etc.) en campo [1] entregan a la salida un valor de tensión proporcional a la magnitud medida, el cual debe ser muestreado, cuantificado y codificado para efectuar el almacenamiento y posterior transmisión de datos a través de la red. Esta función la cumple el convertidor analógico-digital, que es el encargado de realizar la conversión de la tensión de salida de esos sensores.

La elección de la topología de conversión a utilizar se definió a partir de un estudio comparativo entre las posibles opciones aplicando como criterios de selección las restricciones que impone la aplicación [2]:

- Mínimo consumo. A fin de maximizar la duración de las baterías, la reducción del consumo es una de las principales restricciones del diseño.
- Baja tasa de conversión. La baja variabilidad en el tiempo de las magnitudes a medir permite trabajar con una tasa de conversión por unidad de tiempo reducida.
- Resolución de conversión. Tomando como ejemplo típico la medición de temperatura, la precisión necesaria (0,1 ºC en el rango de temperatura ambiente) requiere un conversor con una resolución mínima de 10 bits.

En base a estas restricciones se optó por realizar la conversión mediante un convertidor analógico-digital de aproximaciones sucesivas por reparto de carga. Esta opción puede trabajar sin problemas con la resolución planteada y a la tasa de conversión requerida por la aplicación, mientras que a partir de un estudio preliminar se determinó que su consumo energético es el mínimo posible dentro de las opciones disponibles. [3]

Convertidor de aproximaciones sucesivas (SAR)

El convertidor de aproximaciones sucesivas, cuyo diagrama en bloques se presenta en la figura 3, realiza la conversión bit a bit comenzando desde el más significativo.

En un primer momento se fija el valor del MSB del registro de aproximaciones sucesivas (SAR) a uno, lo que lleva la salida del convertidor digital-analógico (DAC) a un valor de tensión ubicado en la mitad del rango de medición. Si la tensión de entrada es mayor, el bit se deja en 1, de lo contrario se cambia a 0. Este proceso se repite para el resto de los bits del SAR hasta obtener la salida deseada de n bits.

Existe una variante de esta topología pensada para reducir el consumo de energía en la cual las aproximaciones sucesivas se realizan modificando el balance de carga de un banco de capacitores [4], en la figura 2 se muestra el diagrama en bloques correspondiente. El banco de capacitores consta de n+1 capacitores siendo n el número de bits de resolución del convertidor. El primer capacitor tiene un valor C, el segundo un valor C/2, el tercero C/4 y así sucesivamente, hasta los dos últimos, cuyo valor será C/2^{n-1}.

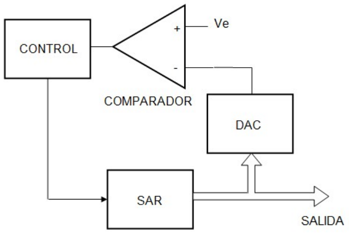


Figura 1. Convertidor de aproximaciones sucesivas.

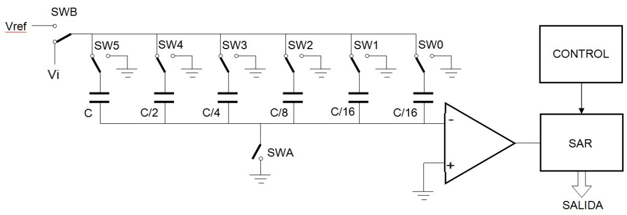


Figura 2. Convertidor SAR de redistribución de carga

Un juego de llaves analógicas permite conectar el borne superior de estos capacitores a la tensión de entrada, a una tensión de referencia o a masa, y la entrada inversora del operacional que coincide con el borne inferior de los capacitores a masa. En la fase inicial de la conversión todos los capacitores se cargan con V_i, luego el primer capacitor se conecta a V_{ref} y el resto a masa, por lo que se constituye un circuito serie entre C y el resto de los capacitores conectados en paralelo. La tensión a la entrada del comparador resultará:

$$V_{entrada\ comparador} = -V_i + V_{ref}/2$$

Dependiendo de si este resultado es positivo o negativo se determina el valor del primer bit, y también se define la posición de la llave del primer capacitor para el siguiente paso. Si el primer bit fue0 el capacitor 1 se conecta a masa, de otra manera se deja conectada a V_{ref}. El proceso continúa conectando el siguiente capacitor (C/2) a la tensión de referencia, con lo que la tensión a la entrada del comparador, por ejemplo, en el caso de que el primer bit haya tenido un valor alto resulta igual a:

$$-V_i + (3/4)V_{ref}$$

Se continúa de esta manera durante n ciclos hasta que se haya determinado el valor de todos los bits.

Dado que todo el nodo se va a implementar en tecnología CMOS como un sistema monochip, resulta imprescindible realizar la mayor cantidad de testeos posibles antes de su implementación para lograr una elevada confiabilidad de funcionamiento. El prototipo desarrollado permite verificar la influencia de los valores de los capacitores, la impedancia de las llaves analógicas y los tiempos de respuesta del resto de los componentes del circuito en la redistribución de la carga del convertidor y sus prestaciones finales.

II. DESARROLLO

El prototipo se implementó utilizando una placa de desarrollo Digilent Netsys con una FPGA Xilinx Spartan 3E para el módulo digital y un circuito con elementos discretos montados en una plaqueta para el módulo analógico a fin de evaluar el impacto de los distintos parámetros de funcionamiento (valor de los capacitores y resistencia de las llaves, frecuencia, retardos, solapamiento de llaves, consumo etc.) en el funcionamiento general, como testeo que complemente la simulaciones previas al diseño como circuito dedicado en tecnología CMOS. Si bien la resolución necesaria para la aplicación prevista es de 10 bits, el convertidor que se implementó en el prototipo tiene una resolución de 2 bits a fin de simplificar el armado de la parte analógica, dado que todos los elementos en estudio toman parte en esta conversión. El diagrama en bloques del conjunto se observa en la figura 3.

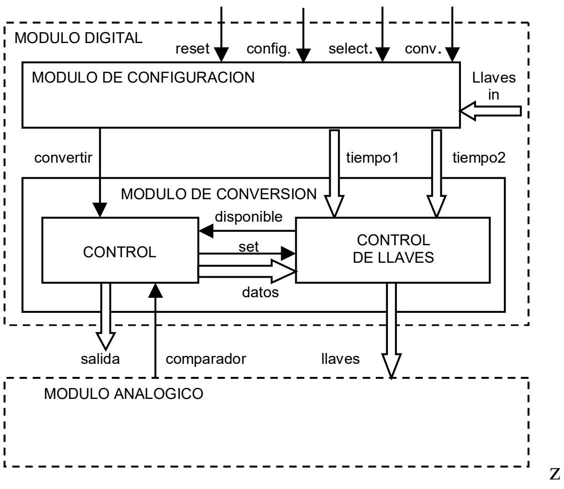


Figura 3. Diagrama en bloques

Módulo Digital

Como se ve en la figura 3, el módulo de control se compone de un bloque de configuración, encargado de permitir la selección de los tiempos de conexión y desconexión de las llaves, y un bloque de conversión que controla la apertura y cierre de las llaves analógicas en la secuencia correcta a partir de la información recibida desde el comparador analógico.

En el inicio el módulo se encuentra en estado de espera hasta que llegue la orden de configurar, en cuyo caso se pasará al estado de configuración, o bien la orden de iniciar el funcionamiento del conversor, dando paso al estado de conversión.

Dado que los conmutadores no pueden implementarse de manera ideal, existirá un tiempo en que todas las llaves estén abiertas en cada conmutación, lo que garantiza que no se modificará la carga almacenada en los capacitores. En el estado de configuración se cargan los registros correspondientes a los tiempos de conexión y desconexión de las llaves analógicas para evaluar su impacto en el funcionamiento general. Al activar la entrada de selección se carga en el registro del tiempo correspondiente el valor seteado mediante las llaves que posee la placa de desarrollo. Para pasar al tiempo siguiente se vuelve a activar la entrada de configuración. De esta manera se evita el efecto del posible rebote mecánico de los pulsadores. Mediante sendos LED se puede conocer en qué estado se encuentra el dispositivo y, en el estado de configuración, cuál es el tiempo cuyo valor se está fijando. En cualquier momento de este proceso en que se active la entrada "convertir", se pasa al estado de conversión con los valores de tiempos que se hayan fijado hasta ese momento.

En el estado de conversión se habilita el funcionamiento del módulo conversor, encargado de generar la secuencia correcta de apertura y cierre de las llaves analógicas de acuerdo al estado de la salida del comparador analógico. Esta secuencia es transferida a través de la señal "datos" de 9 bits al módulo de control de llaves que se encarga de la adecuada temporización de la apertura y cierre de las mismas, a partir de los tiempos fijados en el estado de configuración ya descripto.

Módulo Analógico

Cada valor de capacidad se obtuvo mediante la conexión en paralelo de la cantidad necesaria de capacitores del mismo valor, seleccionados por su valor a fin de compensar las tolerancias individuales y obtener las relaciones de magnitud tan próximas a las deseadas como fuera posible. Si bien inicialmente se utilizaron protoboards para el armado del circuito, por cuestiones de ruido inducido se decidió implementar el circuito analógico soldando los componentes en una plaqueta. Las llaves analógicas se implementaron mediante circuitos integrados CD4066 mientras que para el comparador se utilizó el amplificador operacional de entrada FET TL082, todos de fácil obtención en el mercado local. La tensión de alimentación mínima del operacional no debería ser inferior a ±5V. Por otra parte, el valor de la resistencia serie de las puertas de transmisión que constituyen las llaves analógicas depende fuertemente de la tensión de control aplicada, siendo menor cuanto mayor sea ésta. Para satisfacer estos requisitos se alimentó este bloque con una fuente partida de ±5V, alimentando las llaves analógicas entre +V y –V. Dado que la tensión de alimentación de la FPGA está fija entre 3,3V y masa, se realizó un acoplamiento óptico de las señales que intercambian ambos módulos mediante optoacopladores de dos canales CNY74-2 a fin darle flexibilidad de alimentación a la parte analógica. La tensión de referencia se fijó en 2,5 V para mantener los valores de tensión a la entrada del comparador dentro del rango de tensiones admitido por el operacional. La incorporación de estos optoacopladores limita la frecuencia de uso del prototipo a valores menores a 100 Khz.

Resultados de simulación

En las figuras 4 y 5 se observan los resultados de la simulación del módulo digital. En la primera imagen se observa el proceso de configuración: A partir de la activación de la señal "configurar" el sistema pasa del estado de espera al estado de configuración, en el cual se van fijando sucesivamente los tiempos de apagado y encendido de las llaves analógicas, almacenados en las variables "tiempo_1" y "tiempo_2", mediante sucesivas activaciones de las señales "convertir_in" y "selector_in" que van provocando los cambios de estado "espera_tiempo_1", espera_tiempo_2", etcétera. Una vez realizados estos ajustes, mediante la activación de la señal "convertir_in" el estado cambia a "conversión"

En la segunda imagen se observa el proceso de conversión. El sistema va atravesando sucesivamente los estados de "referencia", "primer_bit" y "segundo_bit". Cada uno de estos estados implica la conexión y desconexión de las correspondientes llaves analógicas a partir de la información contenida en la señal "datos". Esta señal es enviada al módulo de control de llaves mediante un proceso de handshake que envía una señal "set" cuando el módulo de control de llaves se encuentra disponible. Al recibir esta señal el módulo baja la señal "disponible" y procesa la información recibida con la temporización que fue fijada en el proceso de configuración, lo que se ve en los distintos estados "retardo" y "configuración" de la señal "estado_control_llaves". Una vez realizado este proceso se vuelve a activar la señal "disponible" y el módulo de control envía el siguiente dato.

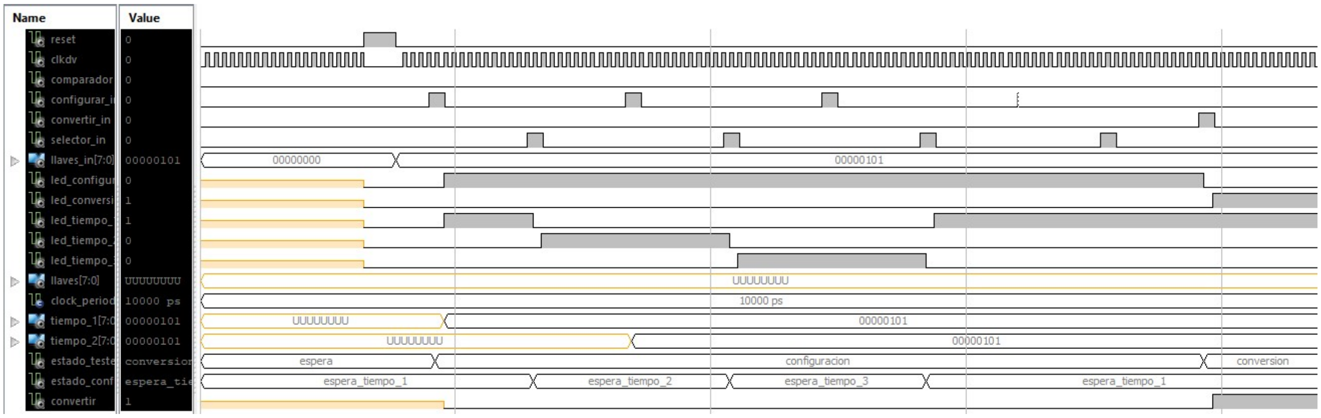


Figura 4. Simulación de la configuración

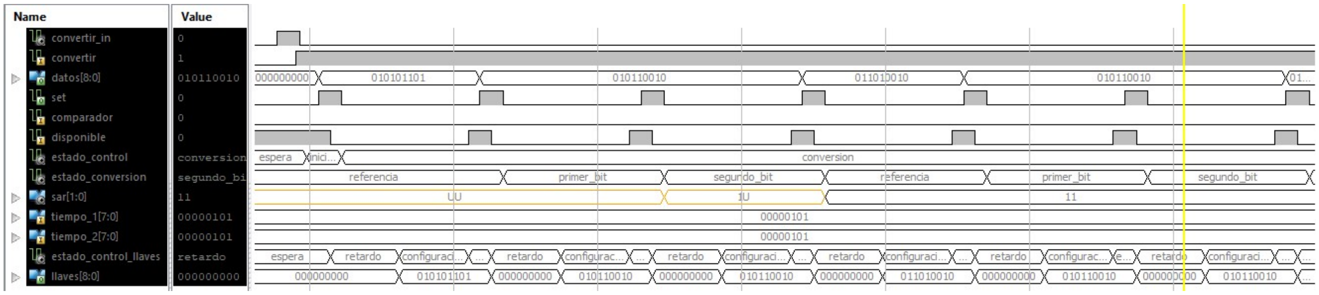


Figura 5. Simulación de la conversión

En cada uno de estos intercambios se va actualizando el bit correspondiente del registro de aproximaciones sucesivas “sar” a partir de la información enviada por la salida del comparador desde la parte analógica. Este proceso continúa indefinidamente hasta que se pulse nuevamente la señal “configuración”

Resultados de ensayos

En las figuras 6 y 7 se observa la forma de onda de la tensión en la entrada del comparador para dos combinaciones de tiempos de conexión y desconexión distintas. Se puede observar el cambio de la frecuencia de conmutación de los capacitores. Las figuras 8 y 9 muestran el equipo bajo testeo y un detalle del equipo.

III. Conclusiones

Se desarrolló e implementó un prototipo dedicado al estudio de la temporización de un convertidor analógico-digital de aproximaciones sucesivas por reparto de carga. Para la implementación del módulo de control se utilizó una placa de desarrollo que contiene un dispositivo Spartan3E500 de Xilinx. Los componentes del módulo analógico (banco de capacitores, llaves conmutadoras y circuito comparador) se implementaron mediante circuitos integrados comerciales y elementos discretos de fácil acceso en el mercado local. Para permitir mayor libertad en cuanto a las tensiones de alimentación se estableció un acoplamiento óptico entre ambos módulos.

Con este prototipo se pretende evaluar el impacto de los parámetros que influyen en el funcionamiento general (valor de los capacitores y resistencia de las llaves, tensiones de alimentación, frecuencia, retardos, solapamiento de llaves, etc.) y, especialmente, en el consumo del circuito.

IV. Referencias

[1] M. I. Schiavon, D. Crepaldo, C. Varela “Control para convertidor analógico-digital de aproximaciones sucesivas” 10th Southern Conference on Programmable Logic (SPL´19), Buenos Aires, Argentina. Abril 2019

[2] .M. I. Schiavon, D. Crepaldo, “Autonomous wireless intelligent network accessible via IP” 7th Southern Conference on Programmable Logic (SPL´11), Córdoba, Argentina. Abril 2011.

[3] M. I. Schiavon, D. Crepaldo, C. Varela “Análisis comparativo de topologías de convertidores analógico-digitales.” IX Congreso de Microelectrónica Aplicada (UEA2018). Universidad Nacional de Catamarca, Catamarca, Argentina, 2018.

[4] Renesas Application note R14AN0001EU0100, “Analog-to-Digital Converters. Operation of a SAR-ADC Based on Charge Redistribution”. Sept. 2020

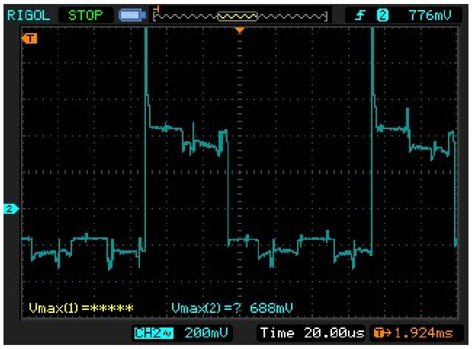


Figura 6. Tensión a la entrada del comparador (caso 1)

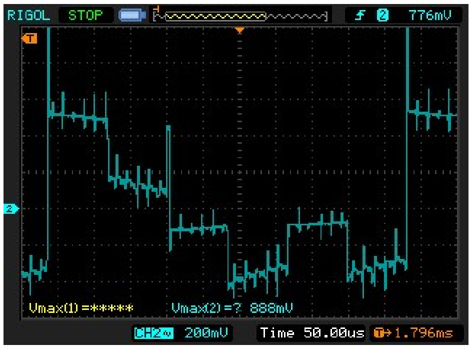


Figura 7. Tensión a la entrada del comparador (caso 2)

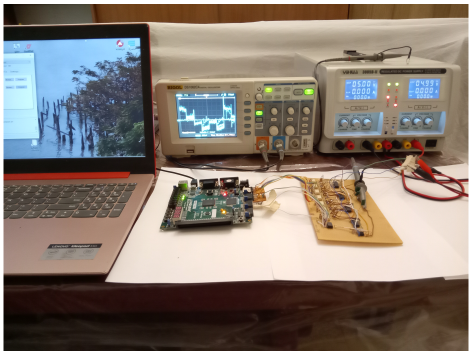


Figura 8. Testeo del prototipo

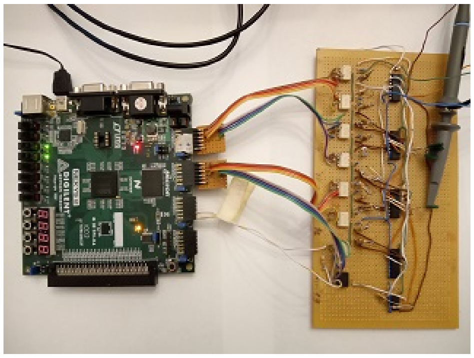


Figura 9. Detalle del prototipo

SHA-3 Implementation for Post-Quantum Cryptography using High-Level Synthesis

Fernando Aparicio Urbano-Molano and Jaime Velasco-Medina
Bionanoelectronics Research Group
Universidad del Valle
Cali, Colombia
{fernando.urbano, jaime.velasco}@correounivalle.edu.co

***Abstract*—This paper presents the implementation of SHA-3 and SHAKE128 using High-Level Synthesis (HLS). The algorithms were initially written in C language and then they were modified and optimized using some directives of the Vivado HLS. Finally, they were synthesized in Vivado on the Virtex-7 XC7VX485T-FFG1157-1, and the Stratix IV EP4SGX230KF40C2 using Intel Quartus Prime Standard Edition. The HLS implementations were compared with those found in the literature, and they were found to have improved frequency, clock cycles, and efficiency. The implementations used an average of ∼2% of LUTs on the Virtex-7 and ∼5.8% of ALUTs on the Stratix-IV. Then, implementations can be embedded into resource-constrained devices for the Internet of Things (IoT) applications. HLS tools allow for efficient design, testing, and implementation of algorithms. Then, HLS-based design is suitable for the hardware implementation of post-quantum cryptography, where complex schemes composed of multiple algorithms can be implemented in a shorter time than using hardware description languages.**

***Index Terms*—SHA-3, SHAKE128, High-Level Synthesis, FPGA, Post-Quantum Cryptography.**

I. Introduction

In December 2016, the National Institute of Standards and Technology (NIST) of the U.S. Department of Commerce initiated the development and standardization process for Public-Key Post-Quantum Cryptography (PQC) [1]. The goal was to achieve PQC systems secure against attacks generated from classic and/or quantum computation. Three Digital Signature Algorithms and one Public-Key Encryption and Key-establishment were currently standardized. Four other PQC candidates advanced to the next round. Some candidates use the Secure Hash Algorithm-3 (SHA-3) family of functions [2] for deterministic random bit generation and matrix generation key, among others.

To the best of our knowledge, the previous works are related to the Keccak and SHA-3 hardware implementations, and none of them is about the extendable-output functions (XOFs), SHAKE128 and SHAKE256. Since the standardization process began and after the publication of the standard, several papers have been published, but few of them use High-Level Synthesis (HLS) design methodology. For example, the authors in [3] presented one of the first FPGA implementations of Keccak-256 and Keccak-512 on Xilinx Virtex-5 FPGA. The authors do not report or explain how they get throughput results in that work.

Other recent works are [4] and [5]. In [4], the authors presented a hardware implementation of SHA-3 - 256 synthesized into the FPGA Intel Arria. The achieved results have shown a high throughput, and in [5], the authors presented a hardware implementation of SHA-3 for the PQC Classic McEliece scheme.

This work aims to use HLS tools to design the SHA-3 – 512 used by the CRYSTAL-Kyber PQC scheme and the SHAKE128, one of the SHA-3 family of functions used by other PQC candidates. In this case, HLS reduces the design efforts and development time; thus, it is a promissory hardware design approach for Internet of Things (IoT) applications. We synthesized the designs on Virtex-7 and Stratix IV FPGA, and we reported experimental results for frequency, area, throughput, and efficiency.

This paper is structured as follows: Section II describes the Keccak and SHA-3 algorithms. Section III presents the HLS-based synthesis and verification approach. Section IV presents the hardware implementation results and comparisons with other recent similar works in terms of area, frequency, throughput, and efficiency. Finally, Section V presents the conclusion of the work.

II. Background

The Keccak algorithm is a hash function designed by Guido Bertoni et al. [6] based on sponge construction. Keccak has seven permutations denoted as Keccak-f$[b]$, where b is the width and can be 25, 50, 100, 200, 400, 800, or 1600-bit. For this work, b is 1600-bit, and $b = r + c$, where c is the capacity of the sponge function, and r is a bit-rate (see Table I). The Keccak-f$[b]$ are structures consisting of a sequence of identical rounds. The number of rounds is given by $nr = 12 + 2l$; where $2^l = b/25$. In this way, $l = 6$, and nr is 24. The Sponge_Keccak $[b](M)$ algorithm takes an input (M) of arbitrary bit-length and performs a hash transformation on an internal state to produce an output (L) of the desired bit-length.

The cryptographic hash functions are called SHA-3 - 224, SHA-3 - 256, SHA-3 - 384, and SHA-3 - 512, and the XOFs are called SHAKE128 and SHAKE256. In hash functions, the input is called the message, and the output is called the

This work was supported by Ministry of Science, Technology, and Innovation of Colombia, and *Universidad del Valle* university.

```
Algorithm 1 Keccak-f[$b$]
Input: $S$ of length $b$; $nr$.
Output: $S'$ of length $b$.
A ← State Array S
for $i = 0$ to $nr - 1$ do
    A ← Round$(A, i)$
end for
$S'$ ← A
return $S'$
```

```
Algorithm 2 Sponge_Keccak[$b$]$(M)$
Input: $M$ and $b$.
Output: $L$.
Padding:
P ← M $\|$ pad[$r$]($\mid M \mid$)
Absorbing:
for $i = 0$ to $len|P|/r - 1$ do
    S ← $S \oplus (Pi \parallel 0^c)$
    S ← $Keccak{-}f[b](S)$
end for
Squeezing:
while $|L|_r r < l$ do
    S ← $Keccak{-}f[b](S)$
    $L \leftarrow L \parallel [S]_r$
end while
```

Table I
PARAMETERS r AND c FOR SHA-3

SHA-3	r	c
SHA-3 - 256	1088	512
SHA-3 - 512	576	1024
SHAKE128	1344	256

(message) digest or hash value. A cryptographic hash function is designed to provide unique properties such as collision resistance and preimage resistance that are important for many information security applications [2].

Algorithm 1 carries out the following steps: 1) The *b-bit* string *S* is converted to a tridimensional state-array *A*; 2) The state-array *A* is computed with the function Round; 3) The last state-array *A* is converted to *b-bit* string *S'* [2].

According to Algorithm 1, the main operation is the Round function, which processes the state using five transformations: χ, θ, ρ, π, and ι, which is the addition of round constants (RC). These transformations are based on simple logic operations: rotations (ROT), AND, XOR, and NOT. The round constants are 64-bit signals represented in hexadecimal form.

As seen before, Keccak is based on sponge construction, which has three stages if the input is included: Padding, Absorbing, and Squeezing.

Algorithm 2 describes the Sponge_Keccak[b](M) function. Where S is the state, and P is the padded message that produces a length multiple of r. This step is conceived to improve the algorithm against length extension attack [7]. The operator $\|$ represents bit-concatenation.

Table I presents the parameters r and c for SHA-3. The standard has specified four output lengths for SHA-3. However, the XOF; SHAKE128 and SHAKE256 can have any length.

Thus, the SHA-3 can be described as:

$$SHA - 3 - c(M) = Sponge\_Keccak[r + c](M \parallel 01) \quad (1)$$

The two XOF functions are defined as:

$$SHAKE128(M, d) = Sponge\_Keccak[256](M \parallel 1111, d) \quad (2)$$

$$SHAKE256(M, d) = Sponge\_Keccak[512](M \parallel 1111, d) \quad (3)$$

Where d is the desired output length.

III. SHA-3 HARDWARE IMPLEMENTATION USING HLS

One disadvantage of FPGA implementation using hardware description languages are its time-consuming design. It is well-known that some implementations in the field of hardware-based cryptography can take several months or even years during the product development process. HLS can facilitate this process because a reference implementation in C of the design or algorithm can be suitable for generating a hardware module automatically or with just one minor modification.

The first step of the HLS-based design process is to use the C reference implementation and evaluate its suitability for HLS synthesis. In most cases, it is required to carry out modifications. The next step is to run a simulation using a C testbench. One of the advantages of HLS tools is that they have directives, known as "pragmas" that allow for better design. The final step is the synthesis that generates an RTL HDL. If the results are not as expected, it is necessary to evaluate what modification or additional *pragmas* are required.

The SHA-3 C code selected for hardware implementation was the improved version written by Daniel J. Bernstein, Peter Schwabe, and Gilles Van Assche[1] and is used for most of the PQC algorithms. The design process was the same for all the implementations.

Our first hardware implementation using HLS was made from the original C code with the main functions (see Fig. 1). To improve performance, we used *pragmas*, including *unroll* to enable some loops iteration to be executed in parallel and *array_partition* to increase throughput. However, the HLS synthesis generated some RAM blocks that were not present in the C code. We addressed this issue by replacing a *constant static* function used in the C code to generate RC constants with a *switch case* inside the *Keccak-f[b]* function. This modification not only reduced the area but also eliminated the RAM blocks.

[1] https://keccak.team/2015/tweetfips202.html

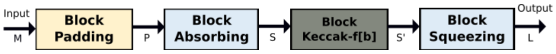


Figure 1. HLS modules.

The absorbing function described in the original C code uses a *static* function that takes an 8-bit array as input and generates a 64-bit output (*load64*) in the *Keccak-f[b]* function. However, since this function is generic, it needs to be made more specific and included in the main function, as follows:

```
for (i = 0; i < r; i++)
    temp[i] = 0;
for (i = 0; i < 17; i++)
    temp[i] = input[i];
temp[i] = 0x1F;
temp[r - 1] |= 128;
for (i = 0; i < r/8; i++)
    s[i] ^= load64(temp + 8 * i);
```

The same was made for the *squeezing* function. A Python script implementing SHA-3 was created to generate a text file with the output, which was then read by a C simulation file to compare it with the obtained results.

After obtaining the initial result, we optimized the hardware implementation using specific C directives for HLS. For instance, we used the *pragma pipeline* directive to improve the performance of some loop iterations, and we also utilized the *apint_set_range* directive, which we defined as *set_range*. These modifications are presented in detail below:

```
for (i = 0; i < 2; i++){
z = set_range(z,15,0,ROL16(input[4*i+0],8));
z = set_range(z,31,16,ROL16(input[4*i+0],8));
z = set_range(z,47,32,ROL16(input[4*i+0],8));
z = set_range(z,63,48,ROL16(input[4*i+0],8));
s[i] ^= z;
}
s[2] = set_range(s[2], 7, 0, 0x1F);
s[20] = set_range(s[20], 63, 63, 1);
```

In the above C code, *ROL16* is defined as an offset (rotation), and the changes in the *squeezing* function are similar. The results improved with these changes (see Results section). In Table II, the SHAKE128 implementations are renamed to facilitate results and comparisons.

A. Throughput and Efficiency

Table II
SHAKE128 IMPLEMENTATIONS

Implementation	Name
I	SHAKE128(128, 128)
II	SHAKE128(144, 163840)
III	SHAKE128(144, 10240)
IV	SHAKE128(76928, 256)

Table III
SYNTHESIS RESULTS FOR IMPLEMENTATION III

Implementation	FFs	LUT	BRAM	Fmax (MHz)	Clock Cycles
Original	12354	37048	5	134.86	3929
Improved	5101	7610	0	135.72	480

The synthesis results used for comparison purposes are frequency (F) and area (A). These parameters allow to deduce the following metrics; 1) throughput, described by equation (4), is calculated using the bit-rate (r) (Table I) and the maximum frequency in MHZ. 2) the number of clock cycles (C) needed to generate the hash value, and 3) efficiency, described by equation (5), which is the relation between the throughput and area.

$$TP = \frac{(r \times F)}{C} \tag{4}$$

$$E = \frac{TP}{A} \tag{5}$$

IV. IMPLEMENTATION RESULTS

In this section, we present the synthesis results of the HLS implementations. We used AMD Xilinx Vivado HLS 2020.1 tools and an FPGA Virtex-7 xc7vx485t-ffg1157-1. In this case, co-simulations were performed to verify the output results. The VHDL generated was synthesized on the same Virtex-7 using Vivado 2021.1 and the Stratix IV EP4SGX230KF40C2 using Intel Quartus Prime 21.1 Standard Edition; the above was possible because we don't use any specific library.

Table III presents the synthesis results for implementation III. The table shows differences between the original implementation and the optimized implementation, where the last one reduces the LUTs by 20%, FFs by 41.29%, clock cycles by 12.21%, and eliminates memory blocks (BRAM). This optimized implementation was used as the basis for the other implementations.

Table IV presents the synthesis results for all HLS implementations. The table shows that they use few area resources, and the frequency is almost the same for all, except for the implementation II on Virtex-7. The first implementation presents good throughput and efficiency, but the others are relatively low in Mbps order. Until the publication of this paper, we could not find works with SHAKE128 results to make comparisons. Almost all implementations use ∼3% of total area utilization on the Virtex-7 and ∼5.8% of total area utilization on the Stratix-IV. For the case of FFs, they use ∼1% on the Virtex-7 and ∼2% on the Stratix-IV.

In this work, we implement the SHA-3 - 512 of the PQC CRYSTALS-Kyber scheme. However, it is not possible to make a fair comparison with this implementation because we could only find one work that used HLS for SHA-3 - 256. In Table V, for computing area and comparison purposes, we use

Table IV
SYNTHESIS RESULTS FOR THE IMPLEMENTATIONS

Implementation	FPGA	FFs	Area/ Utilization (%)	Fmax (MHz)	Clock Cycles	TP (Gbps)	E
I	Virtex-7	3240	6399 LUT / 2	143.31	67	2.87	0.4493
I	Stratix IV	3231	5949 ALUT / 5	148.02	67	2.97	0.4991
II	Virtex-7	5110	7642 LUT / 3	115.19	7206	0.02	0.0028
II	Stratix IV	5098	6308 ALUT / 6	145.07	7206	0.03	0.0043
III	Virtex-7	5101	7610 LUT / 3	135.72	480	0.38	0.0499
III	Stratix IV	5090	6289 ALUT / 6	147.80	480	0.41	0.0658
IV	Virtex-7	6789	9983 LUT / 3	144.40	5837	0.03	0.0033
IV	Stratix IV	6763	8014 ALUT / 8	150.42	5837	0.03	0.0043
SHA-3 - 512	Virtex-7	2597	6234 LUT / 2	162.31	49	1.91	0.3061
SHA-3 - 512	Stratix-IV	2595	5662 ALUT / 4	144.78	49	1.70	0.3006

Table V
SYNTHESIS RESULTS, THROUGHPUT AND EFFICIENCY FOR SHA-3

Ref.	FPGA	Area (Slices)	Fmax (MHz)	Clock Cycles	TP (Gbps)	E
[8]	Zynq-7000	735	118	20000	0.0068	0.000009
[8]	Zynq-7000	1174	124	70	2.97	0.0016
This Work	Virtex-7	1559	162.31	49	1.91	1.22

the relation that one Slice contains four LUTs using vendor information for Virtex-7 [2] . The results show that the efficiency is highly improved compared with others.

V. CONCLUSION

In this paper, we present the hardware implementation of the SHA-3 - 512 for PQC CRYSTALS-Kyber scheme and its SHAKE128 variant using High-Level Synthesis. The original version write in C-language was modified slightly for an initial compilation in the AMD Xilinx Vivado HLS 2020.1. We found the directive *apint_set_range* that allowed the optimization of the implementations, getting good synthesis results with improvements in clock cycles, area, throughput, and efficiency. The VHDL generated was synthesized on Virtex-7 xc7vx485t-ffg1157-1 using Vivado 2021.1 and on the Stratix IV EP4SGX230KF40C2 using Intel Quartus Prime 21.1 Standard Edition.

The synthesis results show that the SHA-3 implementation use few area resources and has a good throughput. They present improvements in frequency, clock cycles, and efficiency compared to others found in the literature using HLS. Our designs used an average of ∼2% of total LUTs available on a Virtex-7 and ∼5.8% of average ALUTs utilization on the Stratix-IV. From the above results, it is possible to conclude that the HLS design is suitable for resource-constrained devices such as the ones used for IoT applications.

Also, these results allowed us to confirm that HLS is a suitable design methodology for fast hardware implementation of cryptography algorithms, achieving minimal time-consuming development.

With this work, we have verified the efficiency of HLS tools and the significance of understanding its directives. This tool will allow the implementation of PQC CRYSTALS-Kyber scheme. To make a fair comparison, it is also necessary to implement the same algorithms in hand-coded VHDL and synthesize them on the same FPGAs.

[2] https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview

ACKNOWLEDGMENT

Fernando Aparicio Urbano-Molano would like to express his gratitude to the Ministry of Science, Technology, and Innovation of Colombia for the scholarship, as well as the AMD University program, the Intel FPGA University program, and the *Universidad del Valle* university.

REFERENCES

[1] “Federal register: Announcing request for nominations for public-key post-quantum cryptographic algorithms.” [Online]. Available: bit.ly/3XxtC8O

[2] N. I. o. S. a. Technology, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” U.S. Department of Commerce, Tech. Rep. Federal Information Processing Standard (FIPS) 202, Aug. 2015. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/202/final

[3] I. San and N. At, “Compact Keccak Hardware Architecture for Data Integrity and Authentication on FPGAs,” *Information Security Journal: A Global Perspective*, vol. 21, no. 5, pp. 231–242, Jan. 2012, publisher: Taylor & Francis _eprint: https://doi.org/10.1080/19393555.2012.660678. [Online]. Available: https://doi.org/10.1080/19393555.2012.660678

[4] A. Sideris, T. Sanida, and M. Dasygenis, “High Throughput Pipelined Implementation of the SHA-3 Cryptoprocessor,” in *2020 32nd International Conference on Microelectronics (ICM)*, Dec. 2020, pp. 1–4.

[5] X. Zhou, L. Wu, and X. Zhang, “Hardware Design of SHA-3 for PQC Classic McEliece,” in *2021 IEEE 15th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, Oct. 2021, pp. 140–144, iSSN: 2163-5056.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Keccak,” in *Advances in Cryptology – EUROCRYPT 2013*, ser. Lecture Notes in Computer Science, T. Johansson and P. Q. Nguyen, Eds. Berlin, Heidelberg: Springer, 2013, pp. 313–314.

[7] S. El Moumni, M. Fettach, and A. Tragha, “High throughput implementation of SHA3 hash algorithm on field programmable gate array (FPGA),” *Microelectronics Journal*, vol. 93, p. 104615, Nov. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026269218308061

[8] H. S. Jacinto, L. Daoud, and N. Rafla, “High level synthesis using vivado HLS for optimizations of SHA-3,” in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2017, pp. 563–566, iSSN: 1558-3899.

A deep learning application for edge-computing device

Roberto Millón*[1], Federico Favaro†[2] and Pablo Ezzatti†[3]

*Departamento de Ciencias Básicas y Tecnológicas, UNdeC
Chilecito (5360), La Rioja, Argentina
[1]rmillon@undec.edu.ar
†Facultad de Ingeniería, UDELAR.
Julio Herrera y Reissig 565, Montevideo, Uruguay
[2]ffavaro@fing.edu.uy
[3]pezzatti@fing.edu.uy

***Abstract*—Deep learning has become the optimal choice within machine learning for solving specific problems, such as natural language recognition and computer vision. However, deep learning models require more powerful computing than other computational tools.In this scenario, FPGAs have shown great acceptance within the available options due to low power consumption and high performance. In addition, new development tools offered by FPGA manufacturers allow high-level synthesis approaches, avoiding hardware-level complications. Instead, managing the development tools to achieve the goals is mandatory. In this study, we design and train a custom CNN to classify handwritten digits of the MNIST database. Then, we compare the performance of the CNN model in two different hardware platforms: an MPSoC ultra96-V2 board and a PC with high-end characteristics. The entry-level MPSoC board classified 10k images in 5.22 seconds, whereas the PC spent 11.64 seconds.**

***Keywords*—Machine Learning, Deep Learning, PYNQ, MNIST, Tensorflow2.**

I. Introduction

Deep learning is a research field within the machine learning area to obtain high-level abstractions in data using hierarchical architectures to imitate how the biological brain perceives and understands information [1]. In 2006, deep learning gained popularity due to the limitations of conventional machine learning techniques for processing raw data, requiring extensive domain expertise and engineering to detect and classify patterns. This technology overcomes the aforementioned limitations by extracting eigenvectors from the data through serially connected nonlinear multi-layer models called deep neural networks (DNN) [2].

Deep learning has become the de-facto approach for numerous application fields, such as natural language recognition, acoustic modeling, financial fraud detection, and computer vision [3]. Within computer vision applications, a particular class of DNN called convolutional neural networks (CNN) are the most widely used, even though they are the networks that demand the highest time and power processing. As CNN become deeper, by increasing the number of layers in their architecture, they require technologies with more computing power than conventional processors (CPUs), such as GPUs, FPGAs, and ASICs -like the TPU offered by Google [4], [5].

Researchers generally choose hybrid approaches for training and deploying DNN. The most common method is to perform offline neural network training with clusters of high-performance CPUs or GPUs and then transfer the model parameters to other technologies for online inference. FPGAs show an increasing acceptance of the available alternatives to carry out network inference due to their high throughput, low latency, and low energy consumption [6]–[9].

Driven by those advantages, in recent years, FPGA manufacturers have developed high-level tools to model neural networks like Xilinx Vitis AI, Intel Open Vino, and Lattice SensAI. These technologies use conventional artificial intelligence frameworks, such as Tensorflow, Caffe, and Pytorch, among others, to model, develop and experiment with neural networks later deployed on FPGAs [10]. In particular, Xilinx Vitis AI includes neural network models, dedicated software cores for deep learning processing (DPU), and libraries for developing intelligence applications. This strategy allows data scientists without knowledge of hardware development to access the benefits of FPGAs.

This work evaluates the use of FPGAs for computing the CNN workflow for inference with a focus on runtime. Specifically, we use the AI paradigm to classify handwritten digits from the MNIST database with the PYNQ 2.7, DPU-PYNQ, and Tensorflow2 frameworks. The FPGA we use for this research is the Avnet ultra96-V2 board, which includes a high-performance and production-ready computing solution based on the Zynq Ultrascale+ multiprocessor system-on-chip (MPSoC) ZU3EG A484. Our focus is on the inline inference stage, so the training stage of a custom CNN model is performed offline, i.e., on a high-level server.

II. IA background

A deep learning model comprises multiple layers of nonlinear processing arranged in series. Data entering the model is passed through the layers to perform various operations until a feature or classification result is obtained in the final layer [11]. Fig. 1 shows the difference between a conventional machine learning model and a deep learning model. In the former, a domain expert extract features from the data, unlike a deep learning model where the net does all the processes.

CNN is the most widely used deep learning approach for computer vision applications; its architecture consists of

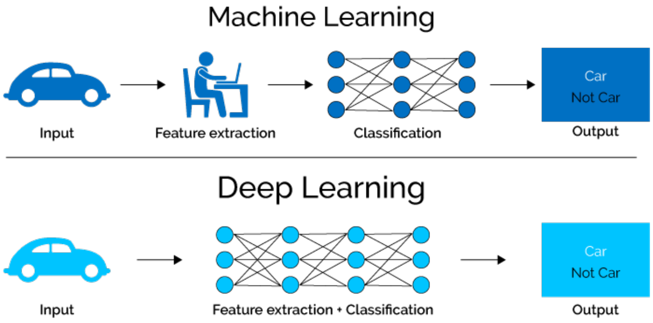


Fig. 1. ML vs DL (Source: softwaretestinghelp.com).

multiple layers trained to minimize an error function. Three types of layers are used in a CNN, as shown in Fig. 2. First, convolutional layers perform convolution operations between the image and different kernels to obtain data features. Second, pooling layers reduce the dimensions of those features. Finally, a fully-connected layer operates like a conventional neural network converting 2D feature matrices into a vector to classify images [1].

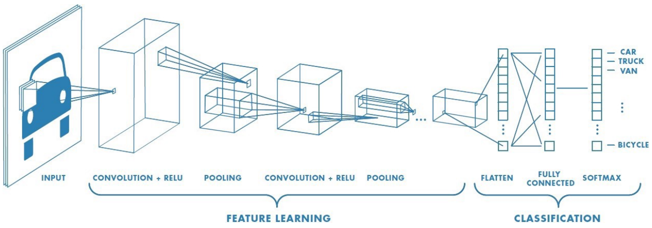


Fig. 2. CNN (Source towardsdatascience.com)

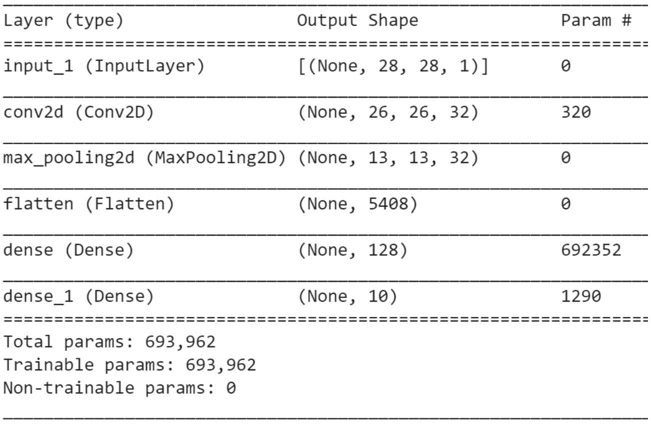
The design and training of deep learning models require an extensive collection of labeled data. With the advance of technology, the amount of data available has dramatically increased, although the data quality is not always sufficient. Because of this, it is common to test deep learning models with established databases such as MNIST. It consists of 70k images of handwritten digits with labels, of which 60k are for training and 10k for testing the neural network. Every image is a two-dimensional 28x28 pixels size in grayscale [12].

Fig. 3. MNIST database (Source medium.com/)

III. Training and Inference of the CNN Model

The workflow of a deep learning project begins with selecting and pre-processing the data of interest. Then, the developer designs and trains the appropriate neural network model for the application. These stages are simplified with Keras and tensorflow2 libraries by incorporating a connection to popular databases such as MNIST, the database used in this work.

The neural network design was carried out within a jupyter environment using the libraries mentioned before. The neural network comprises a convolutional layer integrated by 32 kernels with a RELU activation function, a pooling layer, a hidden layer with a RELU activation function, and an output layer with a softmax function. The model compilation was performed with the adam optimizer and the sparse categorical cross-entropy probability metric. Fig. 4 summarizes the CNN model composition.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 5408)              0
_________________________________________________________________
dense (Dense)                (None, 128)               692352
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 693,962
Trainable params: 693,962
Non-trainable params: 0
_________________________________________________________________
```

Fig. 4. CNN model.

The model was trained with five epochs to obtain optimal accuracy. An additional quantization step was necessary to get a more efficient model for the use of resources. Quantization consists of approximating the 32-bit floating point values of the weights and biases to 8-bit integers, achieving considerable resource reduction without compromising the model's accuracy. At this moment, the model is prepared to be used in an application.

The inference stage began with developing an application in the jupyter environment and loading the dpu.bit, dpu.hwh, and mnist.xmodel files, the last one generated during model training. The application runs on the ultra96-V2 board to classify the MNIST dataset's test data. This work highlights two inference parameters: test accuracy and execution time.

IV. Experimental Evaluation

This section describes the experimental setup and summarizes the results obtained from the evaluation of the inference stage of our CNN.

A. *Experimental Setup and CNN deployment*

The development environment was established in a PC with an Intel Core i7-4870HQ CPU, 16 GB of RAM, and 512 GB of SSD, with Ubuntu 18.04.5 LTS OS, and the Xilinx tools Vitis 2021.1, Xilinx Runtime (XRT) for embedded platforms, and Vitis AI 1.4.916 docker image. In addition, the design and training of the CNN model was executed on the PC. We selected two hardware platforms to infer the neural network: the PC and the MPSoC ultra96-V2 board with PYNQ 2.7 framework.

On the MPSoC board side, we chose the DPU-PYNQ repository that contains the Vitis AI deep learning processor unit (DPU) for deploying CNN and some training and inference notebooks examples ready to run on the device [13].

The following steps were performed to create and deploy the neural network in the ultra96-V2. First, we generated the 4 device-dependent files: dpu.bit, dpu.hwh, dpu.xclbin, and arch.json. These files are mandatory to create the custom CNN model. Second, we executed the Vitis AI docker container and selected the ML framework, in this case, Tensorflow2. The design and development of the CNN model were achieved inside a jupyter notebook.

Finally, the DPU-PYNQ overlay was installed in the ultra96-V2 inside a jupyter notebook. This step configured the DPU core in the PL and connected it to the PS of the MPSoC. With the PYNQ framework, there is no need to struggle with complicated low-level hardware design. Instead, the overlays, or hardware libraries, can be easily configured and imported to the project.

B. Test Cases

The neural network tests were run on the two hardware platforms using the 10k test images from the MNIST database.

C. Experimental Analysis

In our experimental study, we evaluate two hardware platforms to understand the performance of the Ultra96-V2 board in the deep-learning field. Specifically, we developed three variants of the MNIST model. A quantized variant of MNIST leverages the MPSoC board. The other two execute on the PC and correspond to quantized and not quantized MNIST models.

Every CNN version classified the 10K test images of the MNIST dataset in one runtime. We executed five times every variant and informed the average in seconds. The results are presented in Table I

Hardware	Model	Runtime [S]	Accuracy	Size [Kb]
PC	NoquantMNIST	512	0.9866	81
PC	QuantMNIST	11.64	0.9864	23
MPSoC	QuantMNIST	5.22	0.9866	-

TABLE I
MEMORY REQUIREMENTS AND RUNTIME RESULTS

In the first study, we evaluate the consequences of quantization from a quality perspective on both PC models. The model's accuracy is almost the same independently of the quantization step, although there is a 4x model size compression. Considering the runtime performance, the non-quantized model for PC spent 46x more time than the quantized variant for the same hardware. These results highlight the importance of the quantization tool since we can reduce the storage space with a negligible loss of accuracy and achieve a massive improvement in runtime.

On the other hand, the MPSoC version outperforms the CPU quantized version with a speedup higher than 2×. It is essential to highlight that the Ultra96-V2 board is an entry-level, low-cost, and low-power consumption computing board. In contrast, the PC characteristics are better than average.

The previously summarized results show that even using the generic Vitis AI framework to deploy the CNN in FPGA, the reconfigurable device offers important benefits from the runtime perspective.

D. Related works

Previous works show a comparison of CNN models deployed in different hardware. In [14], the authors propose a BNN hardware accelerator design, then implement it on an FPGA, ASIC, CPU, and GPU hardware platforms. The results show that FPGA provides superior efficiency over CPU and GPU but less than ASIC. Another comparative study is proposed in [15] where the author implemented CNN models in two edge devices, an Arty Z7-20 board with an SoC Zynq-7020 (Arm + FPGA) and a Jetson nano GPU board. The results show that the SoC board achieves more performance than the GPU board but implies more effort to deploy the model. [16] is a complete study that provide evaluation of different CNNs and hardware platforms, focusing on the impact of pruning and quantization as optimization techniques. The authors benchmark FPGA, GPU, TPU, and VLIW processors for pruned and quantized neural networks, considering power, latency, and throughput. The result show that FPGAs benefit the most from quantization.

V. CONCLUSIONS

In this work, we have evaluated the use of FPGAs to compute the inferencing stage of CNN. More in detail, we explore the compromise between the easy use of modern direct tools of Xilinx and the achieved performance by the generated configuration by this tool. To reach our objective, we follow an IA workflow for training and inferencing CNN in hybrid architectures. Starting with the mention of the available tools and the step followed to achieve a functional version of an image classification application. We summarize the workflow for training and inferencing neural networks in MPSoC boards. Finally, We evaluated the performance achieved by the low-power and low-budget ultra96-V2 board and compared it with a PC performance. The ultra96-V2 got a higher performance than the PC, spending half the runtime classifying the 10K test images of the MNIST database.

The evolution of FPGA to integrate hybrid technologies with multi-processors like the Xilinx MPSoC used in this work, in addition to new development frameworks that integrate high-level synthesis tools, expands the use of these technologies to researchers without knowledge of hardware development. In particular, the PYNQ framework allows using hardware libraries called overlays without involving low-level details. In addition, conventional IA frameworks like Tensorflow, PyTorch, and Caffe are integrated into the PYNQ tool, improving the development of new IA applications. At this moment, developing IA applications to run in hybrid technologies demands more effort to establish the development environment and understand the conventional IA frameworks than the hardware design complexity described in hardware description languages (HDL). In this scenario, FPGA attributes can benefit embedded software engineers and IA domain experts.

In the future, the team will expand the comparative study in different lines. The most important ones are:

- The extension of the evaluated data set, in particular, including problems with different data types.
- The exploration of new hardware platforms like raspberry pi, Kria KV260 boards, and cloud-computing technologies for datacenters like Alveo cards.
- The inclusion of a deep comparison with a highly tuned design of a CNN for FPGAs.

References

[1] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," vol. 2018, pp. 1–13, 2018. [Online]. Available: https://www.hindawi.com/journals/cin/2018/7068349/

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: http://www.nature.com/articles/nature14539

[3] L. Deng and D. Yu, "Deep learning: Methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014. [Online]. Available: http://dx.doi.org/10.1561/2000000039

[4] D. Ghimire, D. Kil, and S.-h. Kim, "A survey on efficient convolutional neural networks and hardware acceleration," 03 2022.

[5] Y. Tao, R. Ma, M.-L. Shyu, and S.-C. Chen, "Challenges in energy-efficient deep neural network training with fpga," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2020, pp. 1602–1611.

[6] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 93–96.

[7] X. Liu, H. A. Ounifi, A. Gherbi, Y. Lemieux, and W. Li, "A hybrid gpu-fpga-based computing platform for machine learning," *Procedia Computer Science*, vol. 141, pp. 104–111, 2018, the 9th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2018) / The 8th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2018) / Affiliated Workshops. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918318052

[8] K. Seng, P. Lee, and L. Ang, "Embedded intelligence on fpga: Survey, applications and challenges," *Electronics*, vol. 10, p. 895, 04 2021.

[9] Z. Wang, H. Li, X. Yue, and L. Meng, "Briefly analysis about cnn accelerator based on fpga," *Procedia Computer Science*, vol. 202, pp. 277–282, 2022, international Conference on Identification, Information and Knowledge in the internet of Things, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050922005701

[10] Z. Lin, J. M. Ota, J. D. Owens, and P. Muyan-Özcelik, "Benchmarking deep learning frameworks with fpga-suitable models on a traffic sign dataset," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 1197–1203.

[11] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[12] "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[13] "DPU on PYNQ," Dec. 2022, original-date: 2020-04-29T23:43:53Z. [Online]. Available: https://github.com/Xilinx/DPU-PYNQ

[14] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec. 2016, pp. 77–84.

[15] L. Pettersson, "Convolutional neural networks on fpga and gpu on the edge: A comparison," 2020.

[16] M. Blott, N. J. Fraser, G. Gambardella, L. Halder, J. Kath, Z. Neveu, Y. Umuroglu, A. Vasilciuc, M. Leeser, and L. Doyle, "Evaluation of optimized cnns on heterogeneous accelerators using a novel benchmarking approach," *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1654–1669, oct 2021.

Sphery vs. Shapes: A hardware-only raytraced game

1st Victor Suarez Rovere
Buenos Aires, Argentina
suarezvictor@gmail.com

2nd Julian Kemmerer
Philadelphia, USA
julian.v.kemmerer@gmail.com

***Abstract*—In this article we present a tool flow that takes C++ code describing a raytraced game, and produces digital logic that can be implemented in an off-the-shelf FPGA with no use of a hard or soft CPU. We aim for these tools to achieve a friendly C-to-FPGA flow, making the development and simulation process exceptionally fast and easy, while providing high performance and low power results in hardware.**

***Index Terms*—FPGA, raytracing, DSP, simulation**

I. Introduction

In this work we propose a interactive ray tracing system implemented entirely in hardware using a FPGA (Field Programmable Gate Array). It serves as a perfect example that such complex data processing circuits can be developed, tested, and implemented all from a C language based flow. This greatly eases the design process over traditional hardware description languages.

The ability to compile the sources directly as C++ code allows for ultra-fast prototype testing (up to realtime speeds). C++ based cycle accurate emulation tools like Verilator [1] allow for fast simlations, but at a pace that is still much slower than directly compiled C code. A truely realtime simulation workflow is essential for developing an interactive game: to compile-as-C and see the results of code changes executed in realtime is not possible with standard FPGA simulators or synthesis tools.

The project generates each video pixel in hard-realtime "chasing the beam" fashion, without a frame buffer and with negligible jitter. Two medium size FPGA devices were selected for implementation: An AMD™ Artix-7™ (100T) and a Lattice™ ECP5™ (85F). By using many pipeline stages automatically generated by a custom tool, clock rates high enough for playable speeds are possible.

This article is structured with the following sections: hardware platforms and software components are detailed, followed by methods used in the form of a workflow of tools. A final section describes the results obtained in regards to running times, graphics resolution, pipeline depth and power consumption.

II. Materials and methods

This work integrates software and hardware components in a custom workflow that processes C sources through to generating the programmable logic device configuration.

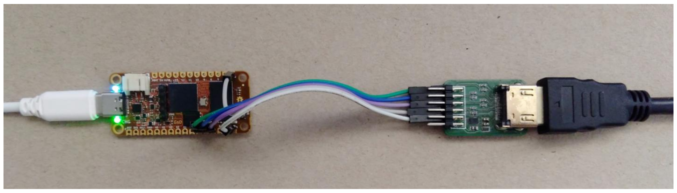


Fig. 1. Hardware prototype.

A. Hardware Architecture

The project was tested on two off-the-shelf FPGA boards:

- A fully open source board based on a Lattice ECP5 FPGA with 85K LE. A Digilent™ PMOD™-compatible accessory was used as a digital video connector to adapt 3.3V levels from the FPGA to CML (Current Mode Logic) compatible levels, suitable for generating DVI (Digital Video Interface) signals capable of driving an HDMI™ display. The main board, a OrangeCrab [2], is shown in Fig. 1 connected to the video adapter.
- A Digilent™ Arty A7 board based on an Artix7-100T FPGA, with the addition of a parallel RGB to DVI adapter.

B. Software Architecture

To get the FPGA bitstream from the C sources, we integrated the following components:

- CflexHDL [3] for C++ parsing, fixed point types and arbitrary width floating point types, and vector of these using operator overloading
- Clang's cindex [4] to help in parsing C++
- PipelineC [5] tool (C to VHDL with autopipelining)
- A custom simulator capable of reading the CPU energy meters to estimate power usage, using the SDL [6] libraries for displaying the rendered graphics
- Yosys [7] tool for Verilog parsing and synthesis and NextPNR [8] tool for place and route (ECP5 target)
- AMD Vivado™ for Verilog to bitstream generation (Artix7 target)
- Verilator for logic level simulation
- GHDL [9] from a Yosys plugin [10] for VHDL to Verilog conversion (used by Verilator and for synthesis)
- LiteX [11] for the SoC design of the test boards, and its video core with serialized digital outputs (DVI)

C. Workflow

The workflow allows writing algorithms involving complex types like structures, fixed and floating point numbers and operations on vectors of these, all composable using a clean and familiar C/C++ syntax.

```
hit_out ray_sphere_intersect(vec3 center,
    point_and_dir hitin)
{
  vec3 rc = hitin.orig - center;
  float b = dot(rc, hitin.dir);
  float c = dot(rc, rc) -
      SPHERE_RADIUS*SPHERE_RADIUS;
  float diff = b*b - c;
  [...]
}
```

As shown in the above function (actual source from the raytracer), the prototype shows a 3D float vector data type is used as 1st argument, and structures are used for the 2nd argument and return type. In the first line of the function's body a subtraction operation is done between 3D float vector types, and in the following line there is a call to a function taking two 3D float vectors and returning a float scalar. This kind of flexibility and clean syntax makes for easy development of math-intensive algorithms, as used in graphics or general DSP (Digital Signal Processing).

The source code is first converted by the CflexHDL tool from C++ to C, translating math operations over types to function calls. Then this subset of C is converted to VHDL by the PipelineC tool. To synthesize that output into a netlist, `ghdl` and `yosys` commands are used, with place and route done by `nextpnr`. Alternatively, the Vivado tool can generate the bitstream right from the generated VHDL files.

In addition to source conversion, the PipelineC tool is primarily the mechanism for producing pipelined digital logic from the pure combinatorial logic derived C code. The tool is aware of the FPGA timing characteristics for the specific device (by iterating with the synthesis tool) and adds pipelining as needed to meet timing. This avoids the tedious and error-prone task of manual pipelining that digital designers are familiar with. The tool reports a preliminary estimate of resources prior to synthesis and the amount of pipeline stages required to implement the user's functionality.

Alternatively, as shown on the left branch of Fig. 2, sources can be compiled and run "as C" as a kind of ultra-fast simulation. As another option, the Verilog sources can be processed by Verilator to generate C++ simulation code which is then run and used to show results graphically in the custom simulation setup we provide. An estimation of resource usage is also reported via instrumented C++ template classes and operator overloading that counts operations between fixed and/or float types. The precision of the fixed and float types can be arbitrarily set simply by changing constants in the source code. Then, using fast simulations, it is easy to iterate on the bit precision number to determine an optimal balance of resource usage v.s. visual graphics quality.

III. Results

We evaluated various aspects of the workflow: how development time is reduced, the maximum resolution achievable using standard video refresh rates, how the PipelineC tool iterated on the number of pipeline stages required to reach the target frequency, and the required precision of data types to meet resource usage requirements on each device (both tested cases). For comparison purposes we also measured the power required by the FPGA platform, and the power usage of a reference PC platform running the game as software.

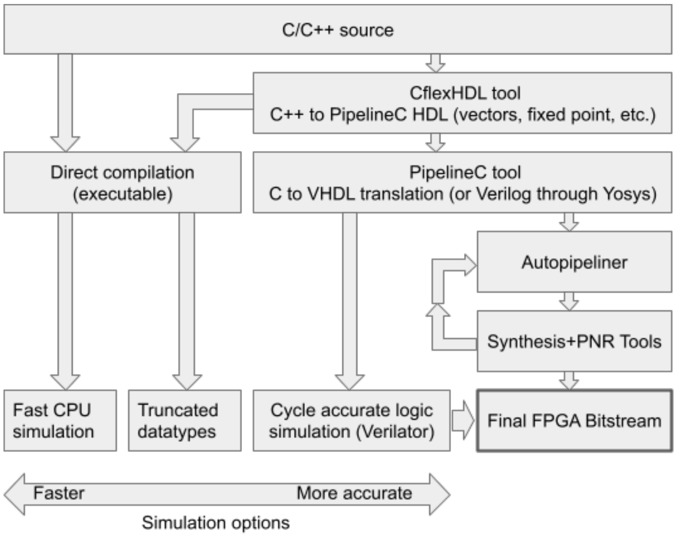


Fig. 2. Tool flow.

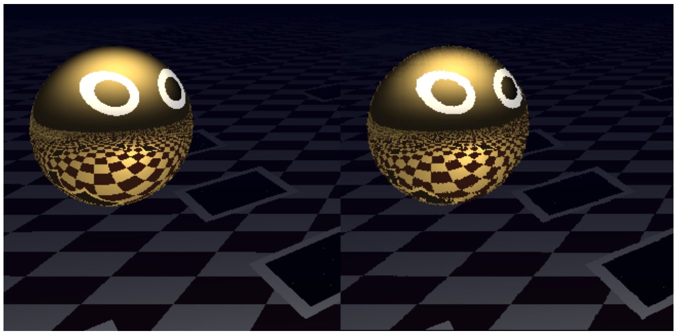


Fig. 3. Effect mantissa precision of float types. Left: 23 bits. Right: 14 bits.

A. Required data types

To be able to fit the design into the the target devices, float types were represented with a 14 bit mantissa (instead of the typical 23 bits). Fixed point processing also needed to save resources, and a total of 22 bits was used: 12 for integer portion, 10 for the fractional bits. Those arbitrary-sized types are provided by the CflexHDL and PipelineC type libraries. The effects of the reduced precision can be readily appreciated with the provided graphical simulation tool, as shown in Fig. 3.

B. Pipeline stages

We selected a target of 148.5 MHz pixel clock on the Artix-7, required to meet standard FullHD video timings (1920x1080 at 60Hz). A 25MHz pixel clock target was set for the smaller

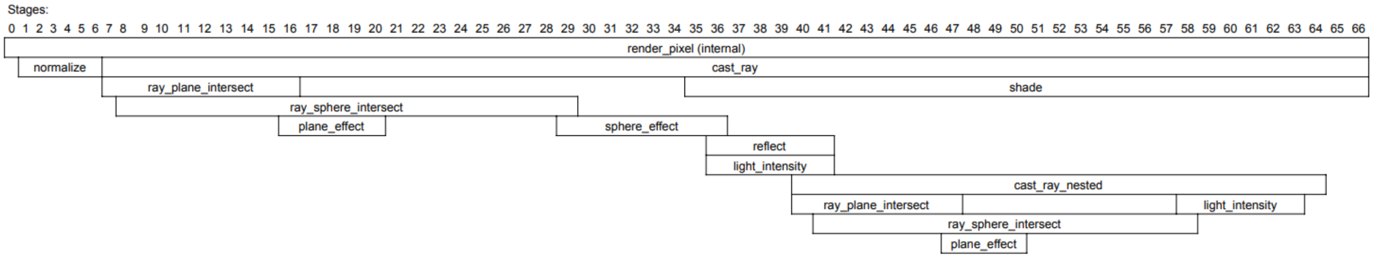


Fig. 4. Resulting pipeline for the ECP5 target.

TABLE I
STAGES PER OPERATION

Operation	Stages
Fixed Compare	1 stage
Fixed Addition/Subtraction	2 stages
Fixed Multiplication	2 stages
Float Compare	2 stages
Float Multiplication	2 stages
Float Addition/Subtraction	3 stages
Float Fast Reciprocal	3 stages
Float Fast Reciprocal Square Root	3 stages
Float Fast Square Root	3 stages
Float Fast Division	4 stages
Float 3D Vector Dot Product	5 stages
Float 3D Vector Normalize	7 stages
Ray Plane Intersection	10 stages
Ray Sphere Intersection	22 stages

TABLE II
TOOL RUNNING TIMES

Simulation kind	Build command	Build time	Speed @1080p
CPU - Fast	make sim	1s	60-86 FPS
CPU - Precise	make gen	5s	40FPS
Logic	make verilator	1min 50s	50s per frame

ECP5, for compatibility with 640x480 pixels at 60Hz. To meet the faster pixel clock on the Artix device, the PipelineC tool created a rendering pipeline of $\sim$400 stages, and $\sim$70 stages were necessary to meet the slower resolution on the ECP5 device. We happily remark that the latter platform allowed for a full open-source toolchain. The resulting pipeline stages for the ECP5 case are shown on Fig. 4. Common operations were subdivided as shown on Table I.

C. Tool running time

The typical times for development/test cycles, as in Table II, are quite fast as compared with the traditional workflows. Build time to run the project during normal devopment is less than a second using a typical PC.

D. Pipelined system compared to a CPU architecture

We compared our design with a quite powerful and modern CPU-based platform: a AMD Ryzen™ 4900H 8-core/16 threads 64-bit CPU @ up to 4.4GHz clock, fabricated using a 7nm silicon process, at 45W TDP (Thermal Design Power), running Linux in a desktop PC. Using the SIMD (Single Instruction Muliple Data) vector float processing offered by the CPU and compiler, we see performance doubled over doing part of the calculations in fixed point (as in the FPGA case). Indeed, the vector extensions and an all-floating point processing was required to reach 60 FPS (frames per second) at the same resolution as in FPGA. The C code for both targets was unchanged beyond simple type definitions to select between float or fixed types (*typedef* C keyword), while the same syntax for math operations was kept by using C++ operator overloading. So the rendering source code remains exactly the same for running interchangeably on the CPU or in the FPGA, even when not all data types are the same.

Comparing a FPGA with a CPU is not an easy task: you can always use a bigger and more modern FPGA than the one we used, but we avoided too large devices on purpose. This is to make more accessible the test platforms, and ease reproduction of our results. Also, you can always use a CPU with more cores, bigger caches, faster clocks, smaller transistors, etc. But each time you add more silicon real estate to a computing platform and - in doing so achieve higher performance - the system will naturally will demand more power. Because of that, we think that evaluating the power consumption per operation is a good way to make a fair comparison.

When running the game on the PC platform, the system uses 97.5% of the CPU (all cores/threads are active). It ran at 88ºC with the fans at their highest speed, and consumed 33W as reported by our simulator which accesses and reports the energy meters internal to the CPU (energy metering was disabled during any excess time after each frame was rendered). The average clock rate of the 16 threads was automatically set by the CPU at 4.220GHz (96% of max clock), thus equivalent to 67.5GHz of a theoretical single-thread CPU. We remark that the power required for the external DDR4 memory bank is not taken into account, nor the energy needed to run fans, but those devices usually take several watts combined.

On the other hand we have our FPGA platform for achieving same resolution and FPS: a medium-sized and low cost chip from same vendor as the CPU, not requiring active cooling, and fabricated on a much older silicon process of 28nm instead of the 7nm of the CPU. We estimated that the FPGA packs just about 13% of the transistor per mm^2 based on average density for those silicon processes [12]. Considering that the FPGA's die size is about half as large (a gross estimation

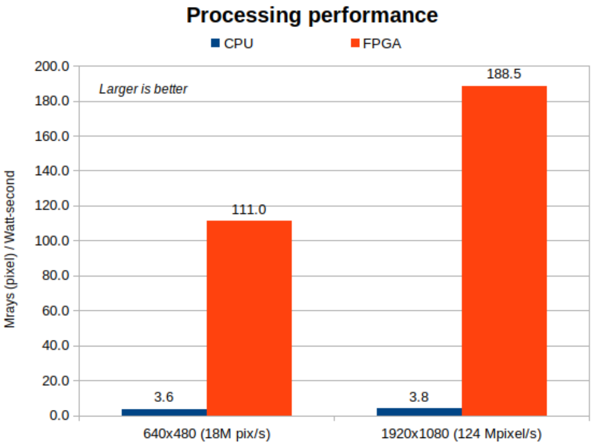


Fig. 5. Processing performance. Blue: CPU. Orange: FPGA

using die pictures having some size references) we calculated the FPGA has ∼15X less transistors and other silicon features compared to the CPU. The clock rate was set to 148.5MHz matching the requirements for the video generation, and the automatically generated pipeline resulted in 482 stages. A total of 135 hardware multipliers are run in parallel, used for floating point or integer/fixed point operations.

The full pipeline required 37Kbits of flip flops, corresponding to an average of 83 bits per operation, constituting a theoretical peak internal bandwidth of 5.4Tbit/s. Computing the 482 stages/∼operations simultaneously at such frequency is roughly equivalent to theoretical 71.5GHz clock rate one operation per cycle execution, a number that closely matches the theoretical single-threaded CPU case.

The workload required to raytrace the FullHD image on the FPGA was on the order of 300 arithmetic and logic operations over integers, and about 150 floating point operations per pixel (including comparisons and optimized operations on just the exponent por power-of-two scalings), all running simultaneously at up to 148.5MHz rate.

Even with all these realtime compute demands, the power required by the FPGA core was just 660mW for the FullHD target resolution, and the chip stayed barely warm. So, our system having an order of magnitude less silicon resources, resulted in 50X less power consumption than a modern CPU running the same workload, or equivalently, a 50X processing performance improvement per unit power, as depicted in Fig. 5. We expect the efficiency gains could improve up to about 6X when using a more modern FPGA (if built on a 7nm process), since Dennard's law seems to hold at the set speed, see [12]. The improvement could be even higher if the digital design were to be implemented in an ASIC, something that we plan to test.

IV. Conclusions

We showed a ready-to-use toolchain for hardware design based on a familiar programming language syntax that greatly accelerates development time by using fast simulators at different stages. The code can be translated to a logic circuit or run on an off-the-shelf CPU. An example application requiring complex processing was demonstrated by writing a game that implements the usual operations for raytracing applications, with a clean syntax for the math and the other algorithms. Since we apply an automatically calculated and possibly long pipeline, the system is capable of performing very well even compared to powerful modern CPUs, but using smaller and embeddable chips, at low power.

The full source code associated with this work can be found on the project's repository [13].

V. About the Authors

This work is a result of the tight interactions between Julian Kemmerer and Victor Suarez Rovere during about a year.

Victor Suarez Rovere is the author of CflexHDL tool used in this project (C++ parser/generator, math types library and simulation) and of the Sphery vs. Shapes raytraced game. He is a software and hardware developer and consultant experienced in Digital Signal Processing, mainly in the medical field. Victor was awarded the first prize in the Argentine National Technology contest, a gold medal from WIPO as "Best young inventor" and some patents related to a multitouch technology based on tomography techniques.

Julian Kemmerer is the author of the PipelineC tool (C-like HDL with auto-pipelining) used in this work. He earned a Masters degree in Computer Engineering from Drexel University in Philadelphia where his work focused on EDA tooling. Julian currently works as an FPGA engineer at an AI focused SDR company called Deepwave Digital. He is a highly experienced digital logic designer looking to increase the usability of programmable devices by moving problems from hardware design into a familiar C language look.

References

[1] Verilator - open source Verilog/SystemVerilog logic simulator - https://www.veripool.org/verilator/
[2] OrangeCrab board - https://1bitsquared.com/products/orangecrab
[3] CflexHDL tool - C to FPGA tool, type library, and fast simulator - https://github.com/suarezvictor/CflexHDL
[4] Clang's cindex parser - https://github.com/llvm-mirror/clang/blob/master/bindings/python/clang/cindex.py
[5] PipelineC tool - C to FPGA with autopipeliner - https://github.com/JulianKemmerer/PipelineC
[6] SDL graphics and UI libraries - https://www.libsdl.org/
[7] Yosys Verilog RTL synthesis tool - https://yosyshq.net/yosys/
[8] NextPNR place and route tool - https://github.com/YosysHQ/nextpnr
[9] GHDL - open source VHDL simulator - http://ghdl.free.fr/
[10] GHDL plugin for Yosys - https://github.com/ghdl/ghdl-yosys-plugin
[11] LiteX system-on-chip creator for FPGA platforms - https://github.com/enjoy-digital/litex
[12] Nadine Collaert, 2016: Device architectures for the 5nm technology node and beyond - https://bjpcjp.github.io/pdfs/chips/SEMICON_Taiwan_2016_collaert.pdf
[13] Sphery vs. Shapes repository - https://github.com/JulianKemmerer/PipelineC-Graphics

Remote Lab: an implementation guide and case study with free hardware boards

Astri Edith Andrada Tivani
Departamento de Electrónica
Facultad de Ciencias Físico Matemáticas y Naturales
Universidad Nacional de San Luis
San Luis, Argentina
aeandrada@unsl.edu.ar

Juan Ignacio Vergés
Departamento de Electrónica
Facultad de Ciencias Físico Matemáticas y Naturales
Universidad Nacional de San Luis
San Luis, Argentina
juaniverges@gmail.com

Julio Daniel Dondo Gazzano
Departamento de Electrónica
Facultad de Ciencias Físico Matemáticas y Naturales
Universidad Nacional de San Luis
San Luis, Argentina
jdondo@unsl.edu.ar

Andrea Schwandt
Department of Electrical Engineering, Mechanical Engineering
Universidad de Ciencias Aplicadas de Bonn-Rhein-Sieg
Sankt Augustin, Alemania
Andrea.Schwandt@h-brs.de

***Abstract*—The article aims to present the conceptual and preliminary results of scientific, pedagogical and technological research oriented to the implementation of remote laboratories. The main characteristics of remote laboratories and a tentative guide for their implementation, based on an international standard, are presented. The use of Python and microcontrollers is a recurrent practice nowadays, which deserves to be taken into account in the creation of a teleoperation project to turn it into a real practice scenario for engineering students. At present, educational institutions are taking into account other independent training processes, mediated by various technologies, in order to promote learning without limitations of location, occupation or age of students.**

Keywords—remote lab, python, microcontrollers, implementation guide

I. Introducción

A principios del siglo XXI se reconoce, en diferentes ámbitos, un cambio de época, a raíz de las transformaciones de las prácticas y códigos comunes, que permean a través de todo el globo. Tal como lo expresa Alcañiz [1] las Tecnologías de la Información y la Comunicación (TIC) son las que mayores modificaciones han experimentado y representan el núcleo axial de los cambios sociales.

Las prácticas de laboratorio, definidas en la siguiente sección, son imprescindibles en la labor educativa, sobre todo de carreras como las ingenierías. Dichas prácticas deben estar accesibles y presentes para usufructuar sus beneficios como herramienta pedagógica en el escenario actual, donde la educación a distancia, la virtualidad y los modelos híbridos delinean los nuevos horizontes de la enseñanza.

En el presente trabajo se proponen los lineamientos para diseñar un laboratorio remoto, enmarcado en la normativa internacional y un ejemplo de aplicación con placas de hardware libre.

II. Laboratorio Remoto

Un laboratorio es tanto el lugar físico donde se encuentra la *planta real* sobre la cual se realizan actividades de experimentación como el proceso de enseñanza-aprendizaje que es facilitado y, a la vez, regulado por el docente que permite integrar los conocimientos relacionados con los conceptos, los procedimientos y las actitudes en la enseñanza y el aprendizaje de las ciencias.

Basándonos en la clasificación de Dormido [2] y en la normativa de la IEEE podemos definir un **laboratorio remoto** como ***un laboratorio de naturaleza física, no virtual, donde el acceso a los recursos es de forma remota para los fines de experimentación.***

La manipulación remota de los recursos de un laboratorio se ha facilitado gracias a la red mundial de Internet, a la evolución de dispositivos audiovisuales (cámaras web, micrófonos, etc), a la evolución de hardware para la adquisición local de datos y al software que permite la sensación de proximidad con el equipamiento.

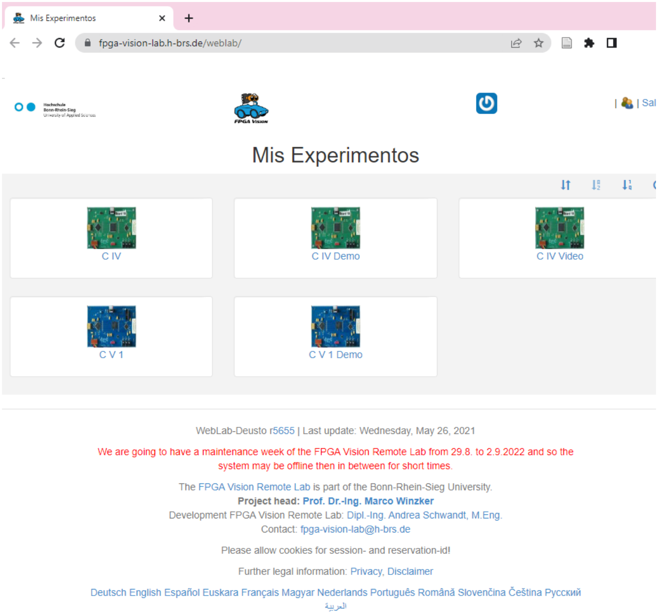
III. Laboratorio Remoto como herramienta en la educación

En la transición de la pedagogía del siglo XX al siglo XXI se ha observado la necesidad de contar con otras herramientas en la educación, como lo ven Romaní y Zaragoza [3], ya que se cambia el centro del "enseñar" al "aprender", sin olvidar que se continúa necesitando del enseñar, actualizando los roles del educador y el educando. Citando textualmente, el aprendizaje requiere estar "mucho más próximo a la "experiencia" de realidad que, a los saberes fragmentados ilustradamente en disciplinas y materias inconexas, por lo menos al entender del que aprende, enmarcando así un reto de altos vuelos y mucho compromiso por parte de todos".

Los laboratorios remotos son una estrategia para el desarrollo del componente práctico en los procesos de enseñanza-aprendizaje, ya que habilitan la optimización de los recursos humanos y materiales de los laboratorios tradicionales, por medio de la integración de las herramientas necesarias para la ejecución de las prácticas, mejorando así la disponibilidad de la infraestructura y equipamiento del laboratorio; a su vez éste tipo de herramienta provee una posible flexibilización de los planes de estudios, fortaleciendo el trabajo colaborativo, el intercambio de ideas y el trabajo en equipo, favorece el desarrollo del autoaprendizaje ya que permite el uso de experimentos a modo de prueba y error, sin miedo a sufrir o provocar un accidente. Se destaca también la reducción de costos de montaje y mantenimiento.

IV. Antecedentes

Para la realización de este trabajo se estudió el *FPGA Vision Remote Lab*[1], proyecto de la Universidad de Ciencias Aplicadas de Bonn-Rhein-Sieg, iniciado en el año 2017. La finalidad del laboratorio es acompañar el aprendizaje del estudiante en la temática del procesamiento de imágenes con una FPGA, posee clases en formato vídeo explicando el algoritmo y la implementación de la detección de carriles, el filtrado FIR y el aprendizaje automático. El hardware real está disponible como laboratorio remoto, las 24 horas todos los días de la semana, para lo cual se le realiza mantenimiento regularmente y, como mínimo, el laboratorio estará disponible hasta el año 2025 La interfaz de usuario puede observarse en la Fig. 1.


Mis Experimentos
fpga-vision-lab.h-brs.de/weblab/
Hochschule Bonn-Rhein-Sieg University of Applied Sciences
FPGA Vision
| Sal
C IV
C IV Demo
C IV Video
C V 1
C V 1 Demo

Mis Experimentos

WebLab-Deusto r5655 | Last update: Wednesday, May 26, 2021

We are going to have a maintenance week of the FPGA Vision Remote Lab from 29.8. to 2.9.2022 and so the system may be offline then in between for short times.

The FPGA Vision Remote Lab is part of the Bonn-Rhein-Sieg University.
Project head: Prof. Dr.-Ing. Marco Winzker
Development FPGA Vision Remote Lab: Dipl.-Ing. Andrea Schwandt, M.Eng.
Contact: fpga-vision-lab@h-brs.de

Please allow cookies for session- and reservation-id!

Further legal information: Privacy, Disclaimer

Deutsch English Español Euskara Français Magyar Nederlands Português Română Slovenčina Čeština Русский العربية

El foco principal de las actividades de laboratorio es cumplir objetivos pedagógicos y desarrollar habilidades experimentales. Por ésto en el año 2019 la IEEE publica el estándar *IEEE Standard for Networked Smart Learning Objects for Online Laboratories* [4] con el propósito de facilitar el diseño, la implementación y el uso de laboratorios en línea para la educación, teniendo en cuenta que las actividades de laboratorio son requeridas en la Educación en Ingeniería, Ciencia y Tecnología.

A. Síntesis del estándar

El estándar define métodos para almacenar, recuperar y acceder a *laboratorios en línea* [2]como a los datos asociados de los denominados *objetos de aprendizaje*[3] los cuales son inteligentes e interactivos.

Para realizar la implementación de laboratorios en línea se utiliza un modelo en capas. La primera capa estandariza un laboratorio en línea como servicio (LaaS), que se puede personalizar en la segunda capa. La segunda capa describe un laboratorio en línea como un objeto de aprendizaje (LO), que se puede integrar en varios entornos de aprendizaje, incluidos cursos masivos abiertos en línea (MOOC), sistemas de gestión de aprendizaje (LMS), repositorios de recursos de aprendizaje y aplicaciones móviles.

En el contexto de este estándar, LaaS se define como un conjunto de requisitos de interfaz que deben cumplirse para satisfacer el primer nivel de estandarización. Esto proporciona una interfaz estándar. Una vez que se establece esta interfaz, un laboratorio en línea debe encajar dentro de los marcos de aprendizaje como un objeto de aprendizaje para proporcionar valor pedagógico. Esta capacidad, sin embargo, no es algo que pueda garantizarse. Por lo tanto, el segundo nivel de estandarización se define como un conjunto de prácticas recomendadas.

V. Lineamientos para implementación de Laboratorios Remotos

A continuación, se propone una guía para la organización requerida al momento de implementar laboratorios remotos:

1) Establecer pautas comunes y describir las acciones requeridas para la implementación de prácticas de laboratorio con acceso remoto entre los interesados.
2) Realizar un manual de implementación de Laboratorios Remotos que cumpla con los requisitos del estándar IEEE Std 1876TM-2019.
3) Establecer las políticas, sistemas, procedimientos e instrucciones con la extensión necesaria para lograr la mejora continua de la calidad del laboratorio remoto a implementar y plasmarlas en el manual confeccionado.
4) Organización y Gestión:

a) *Docente responsable:* cada docente es el responsable del laboratorio que emplaza, por este motivo posee autoridad, recursos e independencia para desempeñar sus funciones. Lo mencionado anteriormente lleva a que el docente esté en condiciones de tomar decisiones relacionadas a los aspectos técnicos y sobre los recursos propios de su laboratorio. Es responsabilidad del docente asegurar que las políticas del manual (del punto 3) se implementen y sean seguidas en todo momento. El docente se debe comprometer a resguardar los datos personales, codificaciones y resultados obtenidos de los estudiantes que utilicen el laboratorio. También es imprescindible que el docente responsable designe un reemplazante en caso de ausencia y éste podrá tomar decisiones en los aspectos relacionados con las actividades del laboratorio. Se requiere la realización de reportes desde la creación, modificación, actualización y otros eventos que se efectúen a raíz de los diferentes cambios del laboratorio remoto.

b) Estructura organizacional: la estructura de la organización educativa a la que pertenece el docente responsable deberá conocer los docentes que poseen laboratorios remotos emplazados y las prácticas que se pueden

[1] https://www.h-brs.de/de/fpga-vision-lab

[2] Un laboratorio al que se puede acceder a través de redes informáticas, como Internet. Un laboratorio en línea puede ya sea virtual, remoto o híbrido de los dos.

[3] Se dice de cualquier entidad digital o no digital que puede ser usada, re-utilizada o referenciada para el aprendizaje soportado en tecnología

desarrollar en los mismos, a fin de distribuir los recursos de forma óptima.

c) *Estudiante: e*l estudiante se debe comprometer a utilizar de forma responsable el laboratorio con las herramientas y pautas brindadas por el docente, para actividades académicas pactadas o de investigación, empleando los tiempos asignados y resguardando la información personal y del laboratorio. El estudiante requiere tomar medidas de seguridad informática personales al momento de hacer uso del laboratorio remoto.

5) Implementación en capas*:* la implementación de los laboratorios remotos responderá a un modelo en capas. En base al estándar IEEE Std 1876TM-2019, se pueden diferenciar cuatro capas, a saber: capa #0 de Hardware y Software, capa #1 Laboratorio en línea como Servicio (Lab as a Service - LaaS), capa #2 describe un Laboratorio en línea como Objeto de Aprendizaje (Lab as Learning Object - LO) y la capa #3 llamado Entorno de Aprendizaje, ejemplos de ellos son los cursos en línea masivos y abiertos (MOOC), los sistemas de gestión del aprendizaje (LMS), los repositorios de recursos de aprendizaje y las aplicaciones móviles. Para observar la figura asociada ver Fig. 2.

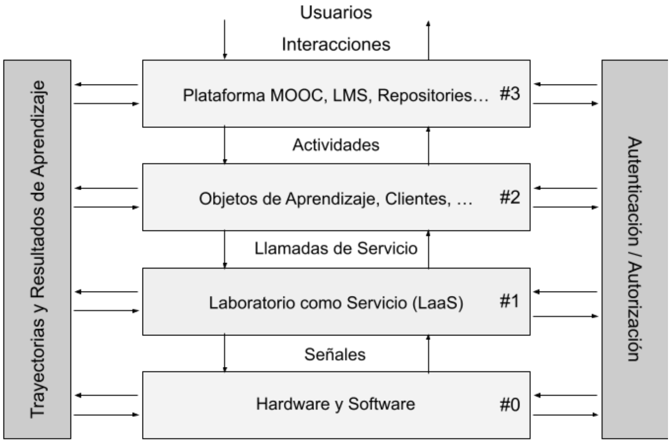

Usuarios
Interacciones
Plataforma MOOC, LMS, Repositorios… #3
Actividades
Objetos de Aprendizaje, Clientes, … #2
Llamadas de Servicio
Laboratorio como Servicio (LaaS) #1
Señales
Hardware y Software #0
Trayectorias y Resultados de Aprendizaje
Autenticación / Autorización

Fig. 2. Capas conceptuales e información del estándar IEEE Std 1876TM-2019

6) Control de la documentación y la información en capas: la distribución y archivo de los documentos, tanto aquellos que regulan el funcionamiento de los laboratorios remotos como los que se generan como consecuencia de su actividad, se realizan en forma controlada por los docentes, permitiendo en todo momento su correcta identificación y trazabilidad. Los documentos deberán estar disponibles en un repositorio digital, donde puedan ser periódicamente examinados, revisados y reemplazados cuando estén obsoletos o invalidados por parte de las personas que los elaboraron.

7) *Revisión de implementación realizada: c*ada laboratorio que se emplaza debe ser revisado para que cumpla con las características esperadas.

8) *Adquisición de suministros:* las compras de insumos que atañen a los laboratorios remotos, si las hubiera, las especificaciones técnicas y el almacenamiento son responsabilidad del docente que emplaza el laboratorio.

VI. Caso de Estudio: Laboratorio Remoto con Placas de hardware libre

Con fines de profundizar el estudio de la implementación de laboratorio remotos se desarrolló un proyecto [5] que permitió a estudiantes acceder a prácticas pactadas, teleoperando hardware emplazado físicamente en la Universidad Nacional de San Luis, a través de una plataforma[4] que está disponible para cualquier navegador web, siendo la única condición indispensable tener acceso a Internet. La estructura utilizada se puede ver en la Fig. 3.

El equipo docente, teniendo como foco principal que los estudiantes puedan desarrollar competencias relacionadas con el diseño de sistemas digitales, definió los objetivos, los conocimientos previos necesarios y las actividades prácticas propuestas. Aspectos tales como equipamiento disponible, tecnología, lenguajes de programación compatibles y factibilidad de reemplazo llevaron a la utilización de la placa de desarrollo NodeMCU ESP32S con el intérprete Micropython.

El manual[5] de la implementación se encuentra en constante actualización, para mantener la premisa de mejorar constantemente la calidad.

La planificación de las prácticas llevó a utilizar hardware adicional, tal como se ve en la Fig. 4, compuesto por diodo LED, display OLED 0.96" SSD1306 y micro servomotor tower pro SG90. La finalidad del mismo es tener actividades de complejidad creciente, manejo de librerías y programación con el lenguaje Python.

La preparación del material[6] para la utilización del laboratorio estuvo a cargo del docente responsable, quien dispuso colocarlo en un repositorio con la digitalización de consignas, material de soporte audiovisual y formularios para evaluar la actividad y la experiencia con el laboratorio.

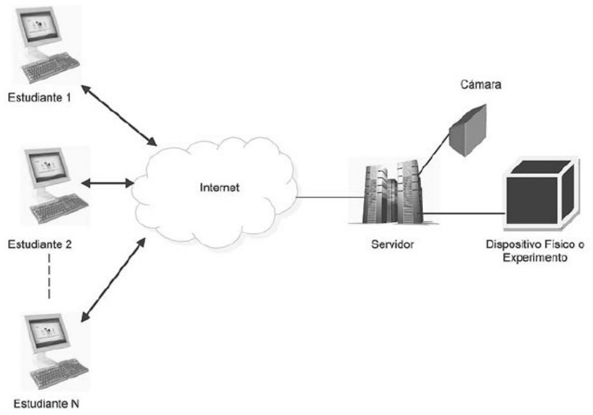


Fig. 3. Estructura de un laboratorio remoto.

[4] La página de inicio de sesión en la plataforma se accede como cualquier página de Internet a través de un navegador web mediante la URL http://labremoto.ddns.net.

[5] El manual está disponible en https://goo.su/OW0ydM

[6] Para obtener el material ingresar a https://goo.su/bykh4

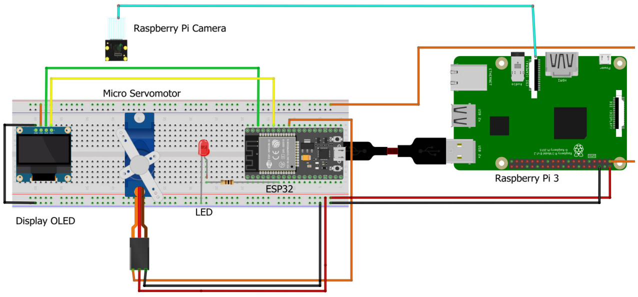


Fig. 4. Conexiones de placa de desarrollo NodeMCU ESP32S, computadora de placa simple Raspberry Pi 3 B y hardware adicional.

Para alojar el servidor y todo el software requerido, aplicaciones, programas y protocolos que responderán a las solicitudes del usuario y grabarán el firmware en la placa ESP32, se escogió la computadora de placa simple Raspberry Pi 3 B. Sumado a esto se le conectó físicamente la placa ESP32 y una cámara, que permitió visualizar la respuesta del hardware luego de la programación realizada.

Previo al uso del laboratorio remoto, el estudiante debía elaborar su código para programar la ESP32. Luego se contactaba con el administrador de la plataforma para que le suministrara las credenciales de acceso. La interfaz de usuario, ver Fig. 5, fue generada con la herramienta Node-RED. En la plataforma se debía ubicar la pestaña de la placa a programar y luego cargar el archivo de extensión ".py" con el código que la placa debía ejecutar. La Raspberry Pi se encargó de que la ESP32 ejecute el archivo subido a través de la combinación de dos aplicaciones: AMPy y RShell, los códigos se ejecutaron a través de AMPy sin que la placa perdiera comunicación con la Raspberry Pi, siempre que la terminal que establecía la conexión de RShell con la ESP32 se mantuviera abierta. Por simplicidad y compatibilidad, la cámara que se utilizó es la Raspberry Pi Camera V2. Dicha cámara en combinación con la aplicación por línea de comandos MJPG-Streamer, capturaron el video y transmitieron en tiempo real la reacción de la placa y del hardware adicional, para que se pueda corroborar la funcionalidad deseada.

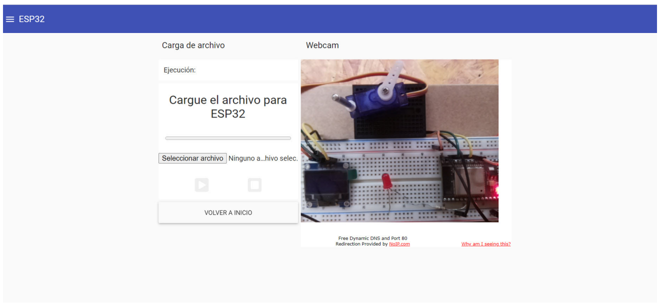


Fig. 5. Interfaz de usuario para programar ESP32

Se realizaron diversas pruebas de cortes de luz, cortes de red en servidor y en usuarios, dos o más usuarios intentando ingresar a la plataforma simultáneamente y otras de acceso desde red externa e interna.

Como resultado de la implementación del laboratorio se observó en sus primeros 3 meses de funcionamiento una utilización del 33,33% sobre un total de 6 prácticas impartidas relacionadas con microcontroladores, en 2 asignaturas de la carrera Ingeniería Electrónica con Orientación en Sistemas Digitales. Una aceptación del 64,28% entre un total de 14 usuarios estudiantes. La apropiación de conocimientos aumentó considerablemente, observando la aprobación de evaluaciones en primera instancia, de 35,71% a 71,42%. Además se utilizó en 2 cursos extracurriculares con una asistencia de 20 estudiantes a cada uno, teniendo una aceptación del 65% en un curso de nivel básico de Python y del 45% en el caso en un curso avanzado.

Conclusiones

Implementar laboratorios remotos suscita mejoras en la enseñanza de las ciencias aplicadas, brindando más oportunidades, ampliando los momentos de experimentación y reduciendo las limitaciones de espacio y tiempo. Es imprescindible, tal como lo indica el estándar, no sólo focalizar en el diseño tecnológico de los laboratorios sino incorporar modelos didácticos y pedagógicos que promuevan la gestión del aprendizaje de manera autónoma por parte de los estudiantes.

Mediante el trabajo realizado se logró tener una guía básica, con capacidad de mejora continua y basada en un estándar internacional, para la implementación de un laboratorio remoto.

El caso de uso confirmó la necesidad de seguir una metodología no sólo del armado de la herramienta pedagógica sino también de su uso por parte de docentes y estudiantes.

En un futuro se espera lograr solventar todos los inconvenientes que afectan a la eficiencia de los laboratorios remotos, tanto en su implementación como en su utilización, para dar un mejor soporte a la enseñanza de la ingeniería, no sólo a nivel local sino internacional.

Referencias

[1] Alcañiz, M. (2007). Cambios, desafíos y riesgos en el siglo XXI. RECERCA. Revista De Pensament I Anàlisi, (7), 5-14. Disponible en: https://www.e-revistes.uji.es/index.php/recerca/article/view/176

[2] Dormido Bencomo, S. (2004). Control learning: present and future, Annual Reviews in Control, 28(1) 115-136, ISSN 1367-5788, https://doi.org/10.1016/j.arcontrol.2003.12.002.

[3] Riera i Romaní, Jordi, & Civís i Zaragoza, Mireia (2008). La pedagogía profesional del siglo XXI. Educación XX1, 11( ),133-154. ISSN: 1139-613X. Disponible en: https://www.redalyc.org/articulo.oa?id=70601107

[4] "IEEE Standard for Networked Smart Learning Objects for Online Laboratories," in IEEE Std 1876-2019 , vol., no., pp.1-57, 30 May 2019, doi: 10.1109/IEEESTD.2019.8723446.

[5] Vergés, J. I. (2022). Laboratorio remoto para manejo de placa de desarrollo ESP32. (RD03 - 747 / 2022) [Proyecto Final de carrera de Ingeniería Electrónica con Orientación en Sistemas Digitales] Universidad Nacional de San Luis. https://drive.google.com/file/d/1Q7pzWmL5oXoAIdN9U7wv5iunuLxL8F3O/view?usp=sharing