This is the accepted version of the following article: A Python-based evaluation framework for stochastic computing circuits on FPGA SoC, 9th International Symposium on Computing and Networking Workshops (CANDARW 2021), pp. 81–86 (11/2021), which has been published in final form at https://doi.org/10.1109/CANDARW53999.2021.00021.
The article was presented at 9th International Workshop on Computer Systems and Architectures (CSA-9), a workshop of CANDAR 2021.

A Python-based evaluation framework for stochastic computing circuits on FPGA SoC

Naoki Fujieda[†]
Department of Electrical and Electronics Engineering, Faculty of Engineering,
Aichi Institute of Technology, Toyota, Aichi, Japan
[†]nfujieda@aitech.ac.jp

***Abstract*—Stochastic computing (SC) has drawn renewed attention from researchers as it can minimize amount of hardware and power consumption. Since a conversion between binary data and bitstreams is required, an SC circuit cannot be simulated or run alone and it spends a lot of time and effort to prepare an evaluation environment. In this paper, we introduce an evaluation framework for an SC circuit on an FPGA SoC, where the circuit is easily integrated into Xilinx's PYNQ platform and its test program can be written in Python. According to our estimation based on an evaluation, an SC circuit with about 600 input and 300 output ports is implementable on a PYNQ-Z1, a low-end development board for the PYNQ platform.**

I. Introduction

Stochastic computing (SC) was first taught by von Neumann in 1950s and it has drawn renewed attention from researchers [6]. In SC, the ratio of '1' in a bitstream of a digital signal is used to represent a number. It is expected to minimize amount of hardware and power consumption of computation circuit because various types of computation can be done with extremely simple components. Major applications of SC include machine learning [7], [10], [14] and image processing [1], [9], where large amount of computation is required but high precision is not necessarily needed.

Although an SC circuit itself might be simple and easily designed, to evaluate it with logic simulation or on a real machine, such as a field-programmable gate array (FPGA), is not a very easy task. Binary input data must be converted to bitstreams and output bitstreams must be converted back to binary. This means an SC circuit cannot be simulated or run alone. It requires some external components, such as stochastic number generators (SNGs) and counters. There is a domain-specific design language for SC circuits called BitSAD [4] and its benchmark applications called BitBench [3]. However, it only gives an SC circuit from a program code written in Scala and such external circuits are not included. Also, to measure the computation error of SC circuit due to its probabilistic behavior, an appropriate test bench is required. Multiple times of tests have to be conducted with different seeds of random numbers. For these reasons, it spends a lot of time and effort to prepare an evaluation environment for SC circuits.

To deal with this problem, we propose an evaluation framework for SC circuits using an FPGA system-on-chip (SoC). Its target platform is the PYNQ platform [17] developed by Xilinx, where an application and a driver for an intellectual property (IP) core on the FPGA can be written in Python. From an SC circuit described in Verilog HDL or SystemVerilog, the framework automatically generates a wrapper circuit and a Python driver for an IP core, reducing the time and effort to evaluate the circuit. Note that we **DO NOT** intend to deploy the core in ultra low-power applications or to argue that PYNQ is a desirable platform for them. The generated IP core is intended to be used in an evaluation purpose only because the main focus of this research is to make an evaluation of SC circuits easier.

The rest of this paper is organized as follows. In Section II, we briefly explain the background of SC. Section III provides the organization of the framework and the design of its IP core, along with the explanation of code generator and Python driver to make use of a user circuit. Section IV presents some examples of the proposed framework. The amount of hardware on the FPGA side is evaluated in Section V. Finally, we conclude the paper in Section VI.

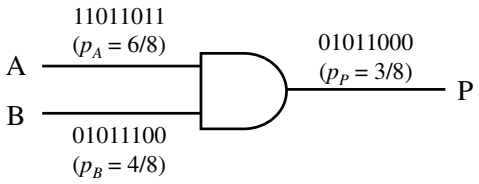


Fig. 1. A simple example of stochastic computing (SC).

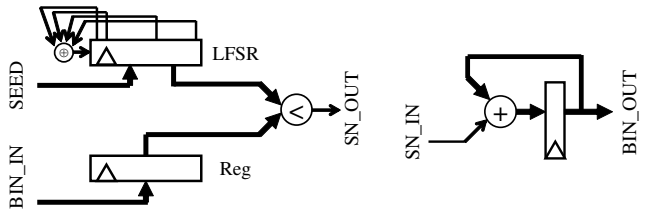


Fig. 2. Typical circuits to convert (a) from binary numbers to stochastic and (b) from stochastic numbers to binary.

II. Stochastic Computing

A. Principles of Stochastic Computing

Figure 1 depicts a simple example of SC. In the most typical representation (unipolar), a value p in the range [0, 1] is encoded to a bitstream, where '1's appear in the probability of p. In the example, the input signals A and B represent the values of $p_A = 6/8$ and $p_B = 4/8$, respectively. Assuming the bitstreams are generated independently from each other, the probability of observing '1' in the both signals is calculated by $p_P = p_A p_B = 3/8$. This means a multiplication in SC can be done with a single 2-input AND gate. Similarly, a product of n inputs can be represented by an n-input AND gate. A number of operators and functions in SC are examined and proposed, such as softmax function [15] and univariate radial basis function [11] in neural networks.

Keeping bitstreams independent from, or uncorrelated to, each other is an important condition for SC circuits to work well. If the inputs are highly correlated, an SC component might express a totally different function from expected. For example in Fig. 1, if A and B have the maximum positive correlation, i.e., '1's in the both signals appears at the same time as often as possible, the corresponding function to the AND gate becomes $\min(p_A, p_B)$. In contrast, some recent studies try to manipulate the correlation to construct functions that would be inefficient in the usual way [9], [16]. Since it is difficult to calculate the effect of correlation precisely, SC circuits have to be tested and evaluated enough using actual bitstreams.

Instead of the unipolar representation, a bipolar representation is used in the case of dealing with negative numbers. There, a value v in the range [-1, 1] is mapped to the probability $p = (v + 1)/2$. For example, from a signal that is assigned $-2/3$ in the bipolar representation, we can observe '1's in the probability of $1/6 = (-2/3 + 1)/2$. It is known that a multiplication in the bipolar SC can be done with an XNOR gate, instead of an AND gate [6]. A *hybrid* design that mixes the unipolar and bipolar signals in the same circuit is also examined [12].

B. Conversion of Stochastic Numbers

Figure 2 describes typical circuits to perform conversion between binary numbers and bitstreams, or stochastic numbers (SNs). A conversion circuit to an SN (Fig. 2 (a)) is often called a stochastic number generator (SNG). The SNG is a comparator between a registered binary input (Reg) and random numbers. A linear feedback shift register (LFSR) is usually used to generate random numbers. The output signal becomes '1' when a random number is smaller than the input number. A new random number is then generated by shuffling the state of the LFSR. This procedure is repeated l times, where l is the length of a bitstream.

Although an LFSR is preferred in the SNG because of its simplicity, any type of random number generator can be used. It includes a *true* random number generator [13], which utilizes an analogue, probabilistic behavior of circuit elements. For example, Knag *et al.* [8] presented an SC architecture where a probabilistic behavior of memristor cells was utilized for SN generation.

Conversion from an SN to a binary number can be done with a counter (Fig. 2 (b)). Since the output binary number is in the range [0, l], the corresponding value in the range [0, 1] is obtained by a division by l.

C. BitSAD and BitBench

In 2019, Daruwalla *et al.* presented a domain-specific design language for SC circuits called BitSAD [4], along with a set of benchmark applications called BitBench [3]. Since BitSAD deals with not only SNs but also audio signals coded by pulse density modulation, they preferred a word *bitstream* computing. BitBench includes a variety of least squares solver kernels and a singular value decomposition kernel as SC applications.

BitSAD is based on the Scala language, which means that a designer can write their own SC circuit along with its test program in Scala. The algorithm can be verified by simply running it as a Scala program. The circuit is generated as a Verilog module, which instantiates library modules corresponding to operators.

In an SC circuit generated by BitSAD, an SN is represented by two signals with postfixes of *_p* and *_m*. The former represents a positive value and the latter represents a negative value. For example, when an SN *A* has a value of $-2/3$, the signal *A_m* has '1's in the probability of $2/3$, while the signal *A_p* is always '0.' We refer to this representation using a pair of signals as a two-line representation in this paper.

Although BitSAD generates a synthesizable Verilog code, it does not provide required external circuits for SC circuits, such as SNGs and counters, which means we cannot test the generated circuit immediately. The authors of BitSAD later added a functionality of emulating SC by software [5], which

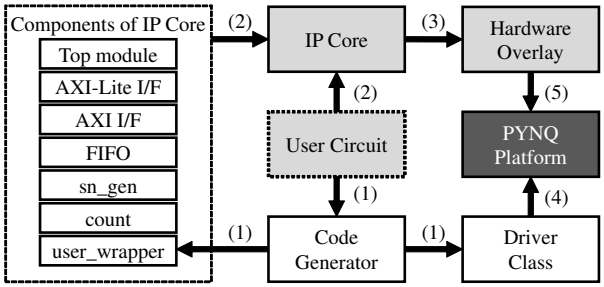


Fig. 3. Organization and workflow of the proposed framework.

might help designers find a failure of SC circuits due to correlation. However, they have not presented how to simulate or run the generated SC circuit, or even whether it works correctly or not.

III. Evaluation Framework

A. Organization of the framework

Figure 3 depicts the organization and the workflow of the proposed evaluation framework for SC circuits. A white box represents a component of the framework. Hardware components (enclosed by a dotted box) are written in either Verilog HDL or SystemVerilog, while software components (i.e., the code generator and the template of driver class) are written in Python. A light gray box is a component provided by a designer or generated by CAD tools. A Verilog testbench is also included, though it is not shown in Fig. 3.

The target of our framework is the PYNQ platform [17], shown in the dark gray box in Fig. 3. PYNQ is a *Py*thon-based system development platform for Xilinx's FPGA SoC called Zy*nq*. There, the FPGA side of the Zynq chip is abstracted as a hardware overlay, which consists of a programming file (.bit) and a hardware hand-off file (.hwh). In a Python program, the FPGA side can be easily programmed from the processor side (called the processing system), by instantiating an `Overlay` class of PYNQ. Also, peripheral circuits in the overlay, packed as IP cores, can be easily controlled using driver classes referenced from the `Overlay` instance.

The workflow of the proposed framework is as follows:

1) From a Verilog HDL or SystemVerilog description of the user circuit, generate a wrapper circuit and a custom driver class based on code templates using the code generator.
2) From the user circuit and hardware components of the framework, package an IP core using the IP packager tool of Xilinx Vivado.
3) Create a Vivado project, make a block diagram with a Zynq processing system and the packaged IP core using the IP integrator tool, and synthesize the block diagram.
4) Upload the hardware overlay and the driver class to PYNQ, and then write a test program in Python.
5) Execute the test program on PYNQ. When the hardware overlay is loaded, the FPGA side will be programmed automatically.

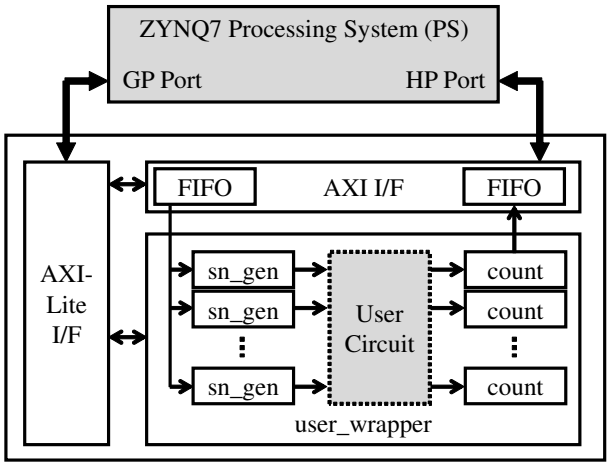


Fig. 4. Block diagram of the IP core of the framework.

Once a designer makes a project and a block diagram in Step 3, they can share them among multiple SC circuits. This is because the interface of the IP core to the processing system is always the same, as we will explain later in Section III.B. If the designer makes another IP core, they will have to change the referenced IP repository directory. Or, if they modify the source files of the IP core directly, they will have to upgrade the core. In the both cases, a new hardware overlay will be obtained by re-synthesizing the block diagram.

B. Design of IP core

Figure 4 describes the internal block diagram of the IP core used in the framework. The core is basically composed of three modules: an AXI-Lite interface (I/F) for control, an AXI interface for data, and the wrapper circuit (user_wrapper). Interconnect circuits are not shown in the figure for simplicity. They will be automatically created between interface modules and ports of the processing system for protocol conversion.

The AXI-Lite interface is connected to a general-purpose (GP) port of the processing system and receives the length of bitstreams and pointers of input and output arrays. The input array is an array of structures, each of which consists of a binary number and a seed of an LFSR, while the output array is an array of counter values. All of them are 32-bit integers. The AXI interface is connected to a high-performance (HP) port and sends read and write requests of the arrays to the processing system. The data are stored in FIFOes. The read data are distributed to SNGs (sn_gen). The data to be written are collected from counters (count) by letting them operate like a shift register.

The wrapper circuit instantiates SNGs, counters, and the user circuit and connects them properly. Since the number of ports and representation of SNs vary with the user circuit, the numbers and configurations of SNGs and counters are defined as parameters. Their values are determined by the code generator, which will be explained in Section III.C. A configuration for each SNG or counter is given as a two-bit parameter. Parameters of `00`, `01`, `10` corresponds to unipolar,

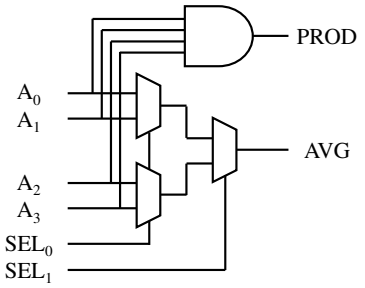


Fig. 5. An SC circuit to calculate the product and the arithmetic mean of four inputs.

```
1  module bit_addmul (
2      input  logic       CLK,
3      input  logic [3:0] A,
4      input  logic [1:0] SEL,
5      output logic       PROD,
6      output logic       AVG);
7      ...
```

Fig. 6. Definition of the module of the SC circuit shown in Fig. 5.

bipolar, and two-line representations, respectively. A designer can define their own representation using the reserved `11` parameter, by modifying the SNG, the counter, and the code generator.

C. Code Generator

The code generator first recognizes the appearance (i.e., the name of module and the names and widths of ports) of the top module of the user circuit, which is essential for instantiation. Clock and reset input ports are recognized (if exist) separately from the others, in order to let them connect to the user circuit directly. It also determines the representation of each SN. If the name of a port has a postfix of *_b*, the corresponding SNG or counter is configured for the bipolar representation. If there are two ports with postfixes of *_p* and *_m*, the corresponding component is configured for the two-line representation (Section II.C).

The generator then outputs source codes of a wrapper circuit and a Python driver from templates. It reads the templates one line at a time. If it is a comment line with a specific keyword, a code corresponding to the user circuit is output. Otherwise, the generator outputs that line without modification.

An example SC circuit is shown in Fig. 5. It has a 4-bit input port A, a 2-bit input port SEL, and 1-bit output ports PROD and AVG. Figure 6 is the definition of the corresponding SystemVerilog module. Note that it includes a clock input for the sake of explanation, even though it is a combinatorial circuit and a clock input is not needed. When the generator recognizes this module, it will detect total of 6 bits of input (except clock) and 2 bits of output. The generated wrapper circuit will thus include six SNGs and two counters.

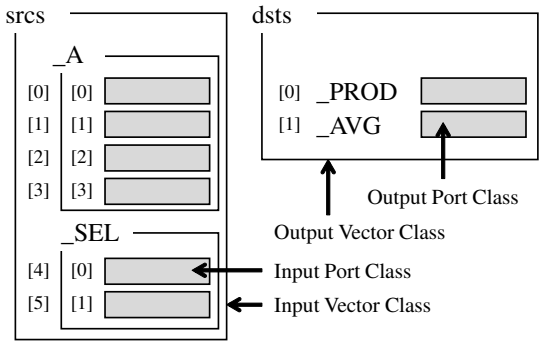


Fig. 7. Organization of I/O class instances corresponding to the SC circuit shown in Fig. 5.

D. Custom driver class

The framework also provides a custom driver class in Python, which inherits the default driver of PYNQ, so that a designer can easily access to the input and output values of the SC circuit. It has instances of four additional classes: input port, input vector, output port, and output vector. A port class corresponds to a single bit of input/output port and has a `value` property to read or write the value of the SN. The input port class additionally has a `seed` property corresponding to the seed of LFSR. A vector class contains multiple instances of a port class and has a `values` property to access them at once.

We explain how the driver organizes I/O class instances, using the example SC circuit (Fig. 5) again. It calculates the product (PROD) and the arithmetic mean (AVG) of four inputs A_0, A_1, A_2, and A_3, by setting the value of SEL_0 and SEL_1 to 0.5. Figure 7 depicts the organization of the corresponding I/O class instances. The driver first makes an instance of port class for each bit of the ports. It then makes two instances of the vector classes, `srcs` and `dsts`, that contains all of the input and output port instances, respectively. Finally, it gives aliases to an port instance or a set of port instances. The name of alias is a prefix of underscore (_) followed by the name of the port. As a result, we can read the value of the product by accessing `_PROD.value` and read out all of the output values from `dsts.values`, for example.

IV. Working Examples

A. Simple SC circuit

As a simple example, we first ran the SC circuit, shown in Fig. 5, on the proposed framework. Figure 8 is a screenshot of Jupyter Notebook running on the PYNQ platform. The box with a label `In [4]` contains a test program and two lines in the bottom with a label `Out[4]` is its outcome.

Since we set $\{A_0, A_1, A_2, A_3\} = \{0.9, 0.8, 0.7, 0.6\}$ in the test program, the expected product and arithmetic mean were 0.3024 and 0.75, respectively. The length of bitstreams was set to 10,000 using the `cycle` property. The SC circuit was ran five times. Before running the circuit, the seeds of LFSRs were reset to random values by calling the `resetseeds()`

```
In [4]: from pynq import Overlay

        pl = Overlay("bitpack_proto.bit")
        core = pl.bitpack_top_0
        core._A.values = [0.9, 0.8, 0.7, 0.6]
        core._SEL.values = [0.5, 0.5]
        core.cycle = 10000

        prods = []
        avgs = []
        result = [['product', prods], ['average', avgs]]
        for i in range(5):
            core.resetseeds()
            core.start()
            prods.append(core._PROD.value)
            avgs.append(core._AVG.value)
        result

Out[4]: [['product', [0.2986, 0.3055, 0.3056, 0.2964, 0.2908]],
         ['average', [0.7482, 0.7449, 0.7524, 0.7507, 0.7445]]]
```

Fig. 8. A test program for the SC circuit shown in Fig. 5 and its execution result.

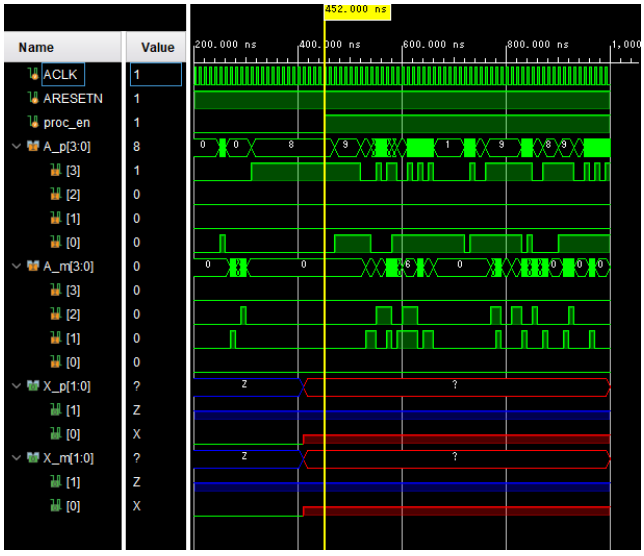


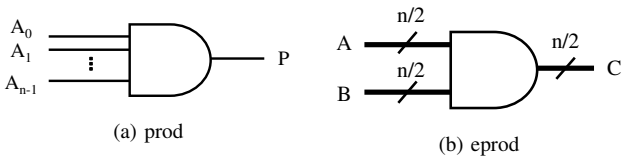
Fig. 9. A part of waveform in the logic simulation of the least squares solver kernel circuit in BitBench.

method. The product and the arithmetic mean for each attempt were recorded in the individual lists and output in the end.

The average of the resultant products and arithmetic means of five attempts in Fig. 8 became 0.2994 and 0.7481, respectively. Though some errors due to randomness were observed, these were similar to the expected values. In this way, we confirmed the operation of an SC circuit using the proposed framework. Also, we greatly reduced the time and effort to prepare the test program.

B. Application in BitBench

Next, we tried to verify the inverse kinematics (IK) variant of the least squares solver kernel in BitBench [3]. We used verilog/ls_ik_top.v file found in the BitBench artifact evaluation dataset [2] as the top module of the user circuit. We confirmed the same file could be obtained by compiling the Scala source with BitSAD. Library modules referenced by the top module were also used. The circuit took two 2×2 matrices



(a) prod (b) eprod

Fig. 10. SC circuits used in the evaluation.

(one of them is the identity matrix), two 2×1 vectors, and two scalar parameters and gave a 2×1 vector $X = [x_0\ x_1]^T$.

The circuit, however, did not work correctly. Concretely speaking, the value of x_0 was far from the expected value and x_1 became always zero. Figure 9 is a part of the result of logic simulation conducted to find out the cause of the incorrect operation.

We confirmed that the input signals were correctly generated by SNGs. For example, we set $\{A_0, A_1, A_2, A_3\} = \{0.7090, -0.2910, -0.2002, 0.5975\}$ in this simulation, which corresponded to the input matrix A. We could observe that, after the reset of the circuit was released (at 452 ns), $A\_p[0]$ and $A\_p[3]$ sometimes became '1' while $A\_m[0]$ and $A\_m[3]$ were always '0.' This was an expected behavior of the two-line representation (see Section II.C) when the corresponding values (i.e., A_0 and A_3) are positive. The opposite held true for A_1 and A_2.

On the other hand, both $X\_p[1]$ and $X\_m[1]$ became hi-Z and both $X\_p[0]$ and $X\_m[0]$ fell into an unknown value (X) during the reset period (at about 400 ns). After we carefully inspected the generated circuit, we found that BitSAD did not properly handle a multiplication by a scalar of a vector or a matrix and it always generated modules of matrix multiplication. As a result, the output became a 1×1 vector instead of 2×1 and x_1 became unused. It also caused a dot-product operation of 1×1 vectors. The dot-product library module did not support such a case and it referred to non-existent signals. That was why x_0 became unknown.

In fact, our initial motivation for developing the proposed framework was to evaluate BitBench in a real machine, but it resulted in finding a bug of BitSAD. Nevertheless, from this example, we can argue that the proposed framework is also a useful tool to help inspect problems of SC circuits.

V. Evaluation

A. Methodology

To evaluate the extendibility of the proposed framework, we synthesize hardware overlays using SC circuits with various numbers of input and output ports. We then check post-implementation reports and record the number of logic elements: look-up tables (LUTs), flip-flops (FFs), and block RAMs (BRAMs), used in the overlay.

In the evaluation, we use two types of SC circuits named *prod* (product) and *eprod* (element-wise product), as shown in Fig. 10. The *prod* circuit is an n-input AND gate, corresponding to a multiplication of n inputs. The *eprod* circuit is an $(n/2)$-bit, 2-input AND gate, corresponding to a element-wise product of two vectors with a length of $n/2$. The number

TABLE I
THE NUMBER OF LOGIC ELEMENTS USED IN THE HARDWARE OVERLAY AND ITS BREAKDOWN.

	LUT	FF	BRAM
IP Core for SC circuit	866	778	2.5
AXI-Lite I/F	84	106	0
AXI I/F	590	349	2.5
wrapper circuit	197	323	0
Interconnect for GP port	285	336	0
Interconnect for HP port	666	764	0
Reset Generator	17	33	0
Total	1,833	1,911	2.5

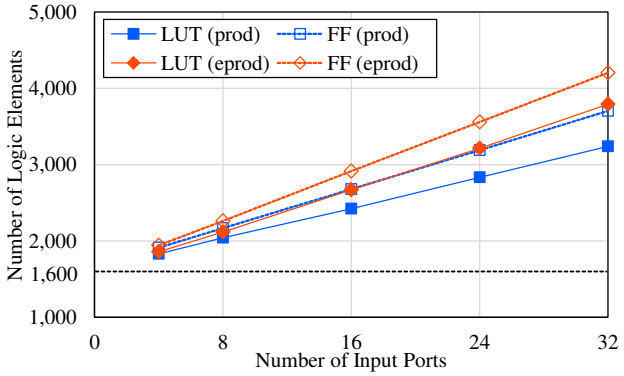


Fig. 11. Relationship between the number of input ports and the number of logic elements.

of input ports is n in the both circuits, while the number of output ports is 1 in *prod* and $n/2$ in *eprod*. We set the number of input ports n to 4, 8, 16, 24, and 32. Note that these user circuits can be negligible in the evaluation results because they are extremely small (taking up only 8 LUTs at a maximum).

Hardware overlays are synthesized by Xilinx Vivado 2021.1 with the default options. The target board is a PYNQ-Z1, one of the low-end development boards for the PYNQ platform, which includes a Zynq XC7Z020 FPGA SoC.

B. Breakdown of the amount of hardware

Table I summarizes the breakdown of the number of logic elements used in the overlay with the 4-input *prod* circuit. Note that the total number of LUTs is not always the same as the sum of those for components, because of logic optimization.

The overlay used 1,833 LUTs, 1,911 flip-flops, and 2.5 block RAMs (0.5 means a half of a 36-kb RAM was in use). More than half of LUTs and flip-flops were used by the AXI interface and the interconnect for the HP port. They are costs of voluntarily accessing the memory of the processing system. All of the BRAMs were used as FIFOes in the AXI interface.

Excluding the wrapper circuit, the number of LUTs and flip-flops became approximately 1,600. This number did not vary widely with SC circuit. Considering that an XC7Z020 FPGA SoC has 53,200 LUTs and 106,400 flip-flops, this is small enough. When we assume that 20% of the rest of elements are reserved for the user SC circuit, we can use 41,280 $(= (53{,}200 - 1{,}600) \times 0.8)$ LUTs and 83,840 $(= 106{,}400 - 1{,}600) \times 0.8)$ flip-flops for the wrapper excluding the user circuit.

C. Effect of the number of ports

Figure 11 plots the number of logic elements used by the evaluated overlays. The X-axis represents the number of input ports, or n, while the Y-axis stands for the number of LUTs or flip-flops. The black dotted line corresponds to 1,600, estimated number of LUTs and flip-flops without the wrapper circuit. We can observe that the number of logic elements increases linearly with the number of ports.

When we let the number of input be i and the number of output ports be o, the formulae $L = 50i + 35o$ and $F = 64i + 32o$ give good approximations of the number of LUTs and flip-flops for the wrapper circuit, respectively. When $o = i/2$, or the same condition as the *eprod* circuits holds, The maximum number of input ports, where the wrapper circuit will fit in the upper limit discussed in Section V.B, was 611 $(\sim 41{,}280/(50 + 35/2))$. Therefore, we can estimate that an SC circuit with about 600 input and 300 output ports can be implemented on a PYNQ-Z1.

VI. CONCLUSION

This paper presented an evaluation framework for SC circuits in order to make them be easily evaluated and verified. We showed its usefulness with some working examples. We also estimated that a relatively large SC circuit was implementable on a low-end development board. The source code of the framework is available at https://github.com/nfproc/PYNQ-BitPack.

As we are going to design and improve our own SC circuits, we expect that the proposed framework greatly accelerates our future studies.

ACKNOWLEDGMENT

A part of this study was supported by the NAGAI Foundation for Science & Technology.

REFERENCES

[1] A. Alaghi *et al.*, "Stochastic Circuits for Real-Time Image-Processing Applications," in *50th Annual Design Automation Conference*, 2013, pp. 136:1–136:6.

[2] K. Daruwalla *et al.* BitBench Artifact Evaluation. [Online]. Available: https://doi.org/10.5281/zenodo.2648959

[3] ——, "BitBench: A Benchmark for Bitstream Computing," in *20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019, pp. 177–187.

[4] ——, "BitSAD: A Domain-Specific Language for Bitstream Computing," in *Unary Computing Workshop held in conjunction with ISCA 2019*, 2019, pp. 2:1–2:5.

[5] ——, "BitSAD v2: Compiler Optimization and Analysis for Bitstream Computing," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 4, pp. 43:1–43:25, 2019.

[6] W. J. Gross and V. C. Gaudet, Eds., *Stochastic Computing: Techniques and Applications*. Springer, 2019.

[7] D. Kim *et al.*, "FPGA Implementation of Convolutional Neural Network Based on Stochastic Computing," in *2017 International Conference on Field Programmable Technology*, 2017, pp. 287–290.

[8] P. Knag *et al.*, "A Native Stochastic Computing Architecture Enabled by Memristors," *IEEE Transactions on Nanotechnology*, vol. 13, no. 2, pp. 283–293, 2014.

[9] V. T. Lee *et al.*, "Correlation Manipulating Circuits for Stochastic Computing," in *2018 Design, Automation & Test in Europe Conference*, 2018, pp. 1417–1422.
[10] B. Li *et al.*, "Neural Network Classifiers Using a Hardware-Based Approximate Activation Function with a Hybrid Stochastic Multiplier," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 15, no. 1, pp. 12:1–12:21, 2019.
[11] V.-T. Nguyen *et al.*, "A Compact and Accuracy-Reconfigurable Univariate RBF Kernel Based on Stochastic Logic," in *2020 IEEE International Symposium on Circuits and Systems*, 2020, pp. 1–5.
[12] K. K. Parhi, "Analysis of Stochastic Logic Circuits in Unipolar, Bipolar and Hybrid Formats," in *2017 IEEE International Symposium on Circuits and Systems*, 2017, pp. 1221–1224.
[13] O. Petura *et al.*, "A survey of AIS-20/31 compliant TRNG cores suitable for FPGA devices," in *26th International Conference on Field Programmable Logic and Applications*, 2016, pp. 1–10.
[14] A. Ren *et al.*, "SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing," in *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 405–418.
[15] S. C. Smithson *et al.*, "Stochastic Computing Can Improve Upon Digital Spiking Neural Networks," in *2016 IEEE International Workshop on Signal Processing Systems*, 2016, pp. 309–314.
[16] D. Wu and J. S. Miguel, "In-Stream Stochastic Division and Square Root via Correlation," in *56th Annual Design Automation Conference*, 2019, pp. 162:1–162:6.
[17] Xilinx Inc. PYNQ: Python Productivity. [Online]. Available: http://www.pynq.io/