Implementing software defined noise generators

Robert Mingesz*
Department of Technical Informatics, Interdisciplinary Excellence Centre, University of Szeged
Szeged, H-6720, Hungary
mingesz@inf.u-szeged.hu

Dénes Faragó
Department of Technical Informatics, Interdisciplinary Excellence Centre, University of Szeged
Szeged, H-6720, Hungary
dfarago@inf.u-szeged.hu

***Abstract*— In the last few decades, the research field related to noise and fluctuations became more and more important and the usage of the results is increasingly promising. In addition to simulations, testing and validation of implemented systems should also be performed on real physical systems. These measurements require noise generators with specific distribution and spectral properties. It is obvious, that the most convenient method to generate such noise is by using digital signal processing. The randomness of these sources is provided by pseudo-random number generators, generators that have limited cycle length/time. Of course, signal generators capable of noise generation are commercially accessible, however, these solutions have limitations. In our previous publication, we presented a Digital Signal Processor based solution capable of generating 1/f^α shaped noises. As a result of advances in technology these days there are many user-friendly and high-performance solutions that provide us new possibilities in noise generation. To ensure proper period length, first, we implemented different pseudo-random number generators on the FPGA based hardware. Then, we adapted and extended our algorithm to the new hardware, as a result, we could significantly increase the sampling frequency, the available bandwidth, the D/A resolution and the number of channels. The noise generator is developed in the LabVIEW programming environment and can run on any FPGA based NI platforms, but it can be ported to any other FPGA based hardware. The new instrument uses a flexible filter arrangement to combine filters of different frequencies to produce more generic spectra up to five decades. The implemented solution is open-source, thus, anyone can use, customize or make further developments of the design for their personal needs.**

Keywords—1/f noise, noise generator, FPGA, DSP, digital filter, software defined instrument

I. Introduction

There are several research fields where noise plays a prominent role. For example, in the case of Fluctuation enhanced sensing (FES), the noise from the sensors carries substantial information about the gases or other substances to be detected [1], in the case of Kirchhoff-Law-Johnson-Noise (KLJN) secure key distribution, noise plays an essential role in encrypted communication [2]. In order to test such systems and to develop the necessary electronic equipment, there is a need for reliable high-performance noise generators that produce an analog signal with appropriate parameters.

Several methods exist for implementing noise generators. If white noise production is required, a combination of Zener or noise-diodes and amplifiers can comfortably deliver superior signal quality up to high frequencies. To produce 1/f^α noises, there are numerous, even completely analog solutions. However, this approach has drawbacks, therefore, in most cases, a mixed-signal solution is chosen where an algorithm produces the desired noise, which is then converted to the desired signal source by digital to analog (D/A) converters. Many of noise generators based on these solutions are commercially available, but most of them have several limitations.

Our approach was to find a solution, that is based on off-the-shelf hardware, thus it can be easily reproduced. Our experiments required an uninterrupted, continuous noise source, with substantially long cycle length. The required sampling frequency was over 100 kHz, up to 1 MHz, and the spectral distribution of the noise had to be more general than a 1/f^α noise.

II. Cycle length and quality of pseudo-random-number generators

True random generators are ideal in many ways to provide sources for random noise generation. However, realizing true random generators is a complicated and expensive task. In addition, noises generated using true random sources cannot be reproduced and the same experiment cannot be performed again. In contrast, algorithm-based pseudorandom number generators can be implemented easily using a small amount of resources. Moreover, the same random sequence can be reproduced by resetting the algorithm with the same beginning state (seed). Compared to the true random generators, their disadvantage lies in their deterministic nature and the finite length of the produced number series.

In today's world, long-term measurements are necessary in numerous cases where a short cycle length can cause unpleasant problems. Commercially available arbitrary waveform generators (AWG) can generate arbitrary signals, but they generate noise with limited and often short cycle length due to finite memory or the weakness of the random generator contained therein.

Colored noise with arbitrary spectral shape can be produced using FFTs. However, the produced signal is definitely finite, and seriously limited by the available memory, currently, it is impractical to create continuous noise with more than 2^{32} points.

Currently, there are many high-quality pseudorandom number generators that can produce long enough cycle lengths that are suitable for most of the applications. If even longer sequences are required, then different random generators can be combined. This method not only increases the cycle length but also increases the independence of the elements in the sequence of numbers, thereby improving the uniformity and quality [3].

While the cycle length of generators is given in samples, in real measurements it is important to compare them to applicable measurement times. A simple comparison is done in TABLE I.

*Corresponding Author

This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 (“Internet of Living Things”)

TABLE I. COMPARISON OF CYCLE LENGTHS WITH APPLICABLE MEASUREMENT TIMES

Cycle length	Sampling frequency		
	100 kHz	***1 MHz***	***10 MHz***
65536 points (16 bit pseudorandom generator)	0.65 s	0.065 s	6.5 ms
8 million points (typical AWG)	80 s	8 s	0.8 s
128 million points (AWG with extended memory)	21 min	2 min	12.8 s
4 billion points (2^{32}) Approximate maximum size of FFT based noise generator	12 hours	1.2 hours	7.2 min
2^{64} points 64 bit LCG or Xorshift pseudorandom number generators	5.9E+6 years	5.9E+5 years	5850 years
2^{128} points 128 bit Xorshift generators	1.1E+26 years	1.1E+25 years	1.1E+24 years

As shown in TABLE I. short cycle noise generators will repeat the generated signal in a very short period of time, which is unacceptable for most of the measurements. It will be even more problematic if we use more than one signal source for our measurement at the same time, the result of the measurements can be simply incorrect. Using 64-bit or 128-bit generators significantly decreases the chance of such artifacts.

The quality of the random generators, the randomness of the generated number series, may be critical in most cases, especially for cryptographic applications. Quality of generators is usually verified by statistical tests. One of the most widespread statistical test collection is the Diehard tests [4][4], with an open source implementation of Dieharder [5]. We tested our implementations using this set of tests, the results are shown in TABLE II.

III. REAL-TIME NOISE GENERATION HARDWARE

Since the cycle length of FFT based noise generators are severely limited for most of the desired applications, the only possibility is to create a continuous noise stream in real time. For this, we first generate a source of random numbers, then we form the spectral shape using a filter bank, then we apply a proper D/A conversion.

While a personal computer has enough processing power to perform this task, since it is not designed to be a real-time system, it is not suitable for continuous noise generation. Additionally, the bandwidth limit between the PC and the D/A card can also cause problems. For this reason, we are looking for devices, that are designed to be real-time systems. The most suitable solutions:

Microcontrollers: currently available 32-bit ARM Cortex M4 and M7 microcontrollers can easily handle 32-bit floating-point numbers, have clock frequencies over 200 MHz, while they also have multiple built-in D/A converters with sampling frequencies over 1 MHz.

Digital signal processors (DSP)**:** these devices are especially designed for real-time signal processing; our previous noise generator was based on an Analog Devices DSP [6]. These days, more and more DSP functionalities are built in high-performance microcontrollers or FPGAs.

Field-programmable gate arrays (FPGA)**:** these devices have a large number of logic gates suitable for implementing high-speed digital devices. Modern FPGAs also contain a high number of DPS slices. For this reason, it is especially suitable to generate multiple streams of real-time noise data.

Our goal was to find an off-the-shelf solution, that can provide enough processing power at an affordable cost. We implemented our solution based on a National Instrument FPGA hardware (NI USB7856R) [7]. The hardware can be conveniently programmed in LabVIEW programming environment, the performance of the hardware implementation can be easily compared to simulation and theoretical results.

The NI USB7856R is a USB based instrument containing a Kintex-7 160T FPGA. It has eight 16-bit D/A converters, capable of sampling frequencies up to 1 MHz. The FPGA has 101,400 LUTs and 600 DSP slices. The maximum clock frequency is 200 MHz, in our application we used a 40 MHz clock. Our solution is compatible with other types of NI multifunction reconfigurable I/O devices (USB, PCIe, and PXIe based) and cRIO devices.

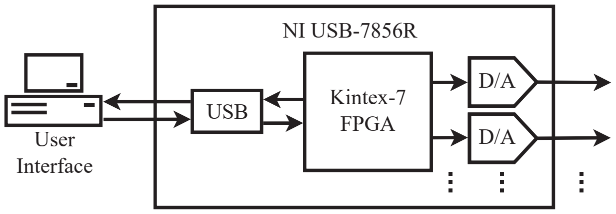

Fig. 1. Block diagram of the noise generator hardware

IV. IMPLEMENTING RANDOM NUMBER GENERATORS

There are many well-known pseudo-random generators, we selected the following ones to be implemented on our hardware: Xorshift [8], Linear congruential generators (LCG) [9], linear feedback shift registers [10], and Mersenne Twister [11]. We implemented these generators as LabVIEW code, this solution provides a handful of possibilities of using the generators in different, LabVIEW compatible FPGA based platforms, as well as in other real-time systems or on traditional personal computers.

Most of the random generators can be implemented in different word lengths resulting in different cycle lengths. The generators were tested using the previously mentioned Dieharder tests, the results, as well as the resource usages are summarized in TABLE II.

The LCGs, while widely popular, are not the best performing generators. Their bad quality is mostly caused by their hyperstructure as well as the shorter cycle length of their least significant bits [9]. However, in our application, since we are using only the most significant bits, this may not cause significant problems.

The linear feedback shift register may seem a good solution considering its cycle length and resource usage; however, it requires multiple cycles in order to generate a single random number.

The Xorshift generator (Fig. 2) is generally a high-quality generator with limited resource usage.

Implementing the Mersenne Twister is not so straightforward; at the same time, it passes all the tests and has an astonishingly long cycle length. A long initialization sequence is also needed, but since it has to be done only once, it can be implemented on the host computer sparing a lot of valuable FPGA resources.

Note: none of the implemented random generators are cryptographically secure, for this reason, they should not be used in live cryptographic applications.

TABLE II. QUALITY, CYCLE LENGTH AND RESOURCE REQUIREMENTS OF IMPLEMENTED PSEUDORANDOM NUMBER GENERATORS

Pseudorandom Number Generator	**Quality and Cycle Length**		
	Passed Diehard Tests	***Maximum Cycle Length***	***FPGA resource utilization***
Linear congruential generator	3/17	2^{64}	Slices: 812 DSP48s: 10
Linear feedback shift register	12/17	2^{160}-1	Slices: 716 DSP48s: 0
Xorshift	15/17	2^{128}-1	Slices: 737 DSP48s: 0
Mersenne Twister	17/17	2^{19937}-1	Slices: 566 DSP48s: 0

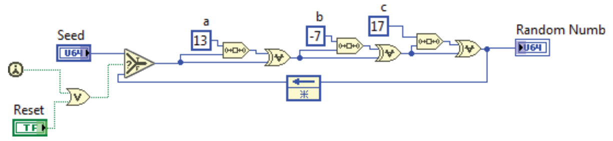

Fig. 2. Labview implementation of the 64bit Xorshift pseudoranom number generator.

All the implemented random generators produce a uniformly distributed number. When using noises, we generally need a Gaussian-distributed noise. To achieve this distribution, we can use different algorithms. One of them is the well-known Box-Muller transform. The main drawback of this method is that it requires the usage of floating-point arithmetic and several functions that are hard to be implemented on FPGAs. Another method to approximate Gaussian distribution is to sum a number of uniformly distributed noises. In our implementation, we summed 12 random numbers to produce a single normally-distributed number.

V. ACHIEVING THE DESIRED SPECTRAL SHAPE

In the previous implementation, we used first-order low-pass filters to generate $1/f^{\alpha}$ noises. However, to perform a wide variety of experiments, it is indispensable to generate more generic spectra. Using strictly low-pass filters limits the possibilities, therefore we implemented generic components that can serve as high-pass, low-pass or band-pass filters. These are composed of two traditional IIR filters that are connected in series. The desired spectral shape can be achieved by applying a bank of these filters on the Gaussian random number source, as visualized in Fig. 3.

These filters can be easily parameterized by their cut-off frequencies and amplitudes. In order to achieve the desired spectrum, these parameters must be optimized. In our setup, the cut-off frequencies are first equally distributed over the relevant frequency band. Then we can select which type of filters do we want to use: low-pass, high-pass or band-pass. In the next step, we provide a set of starting values for filter amplitudes. The LabVIEW's built-in differential evolution-based global optimization function [12] can be used to optimize only the filter amplitudes or both the amplitudes and cut-off frequencies.

The solution uses only the analytically calculated transfer functions. The goodness function of the fit is the absolute difference between the desired and the realized power spectra in the selected frequency range. In our previous solution, we selected a more complex function, of course, there is also the possibility to consider other aspects here.

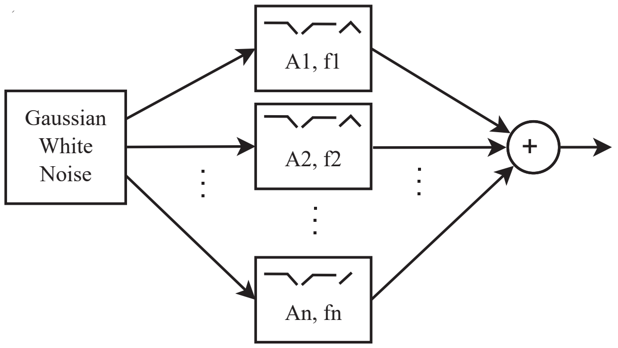

Fig. 3. Obtaining the desired spectral shape using a bank of generic filters (low-pass, high-pass or band-pass)

The optimization is done on the host PC and may take several minutes, depending on the number of filters and the desired shape. During the optimization, we can consider the sin(x)/x effect of the D/A conversion.

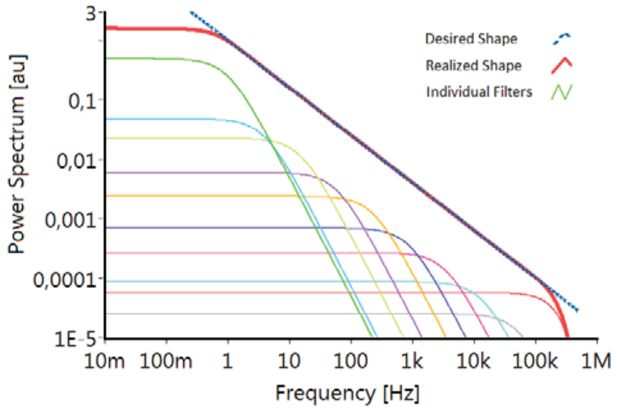

Fig. 4. Filter setup for achieving a $1/f^{0.8}$ noise.

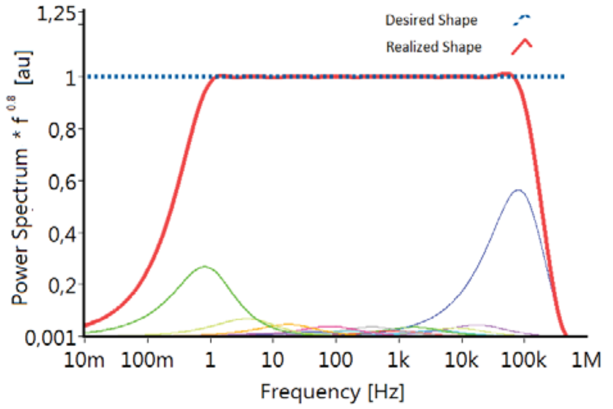

Fig. 5. Filter transfer multiplied by $f^{0.8}$ to emphasize the difference between the desired and realized filter.

In Fig. 4 we can see the result of such an optimization when the desired output was a pink noise with α=0.8 using only low-pass filters. In the next figure, we multiplied the spectrum with $f^{0.8}$ to emphasize the difference between the desired and realized filter. We can observe that the filters with most significant roles are the first and last one.

VI. Performance analysis

In our hardware tests, we used a 128-bit Xorshift pseudo-random generator with a filter bank containing 10 filters. To save DSP resources, a single filter component implemented on the FPGA fabric is used to sequentially calculate the result of the filter bank. This filter, depending on filter coefficients, can serve as a first or second order low-pass, high-pass or band-pass filter. The used hardware has a maximum sampling frequency of 1 MHz, the implementation was fast enough to provide the required data rate. Higher filter count or higher data rate can be achieved by increasing the parallelism in the hardware. The resource usage on the NI USB-7856R is summarized in TABLE III. Based on these results, we can observe that up to 6 parallel noise generators can be implemented without further optimization.

TABLE III. Device utilization of our current setup on the NI USB-7856R

Resource	FPGA Resource Utilization	
	Percent	*Quantity*
Total Slices	16.1%	4084 out of 25350
Slice Registers	9.2%	18610 out of 202800
Slice LUTs	7%	7091 out of 101400
Block RAMs	0.3%	1 out of 325
DSP48s	5.8%	35 out of 600

We extensively tested the implementation using software simulations and real measurements. As a demonstration, in Fig. 6 we generated a signal that has a corner in the spectrum at 13 kHz. In Fig. 7 we demonstrate the measured spectrum of the generated analog signal.

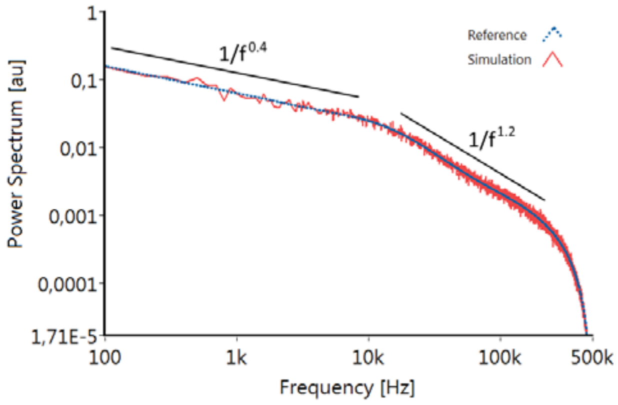

Fig. 6. Demonstration of a specially designed spectral shape containing regions with different exponents. The spectrum of hardware-simulated signal is compared to the desired reference shape.

VII. Conclusion

We realized an efficient mixed-signal noise generation system based on the widely known method of applying a filter bank to modify the shape of a noise. The implementation can support up to six independent noise outputs at a 1 MHz sample rate. The parameters of filter banks are set precisely by an optimization algorithm, resulting in spectral shapes that could not be achieved otherwise.

The implemented software is open source and is created in the widespread LabVIEW programming environment. The code can be used on a wide range of NI FPGA devices, and beside the implemented algorithm it can be easily supplemented with further unique functions, which provide new possibilities like a mixture of noise and deterministic signals, synchronous excitation, and measurement.

Further information and the source code are available on our webpage[13].

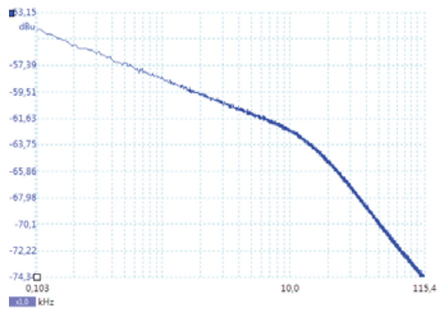

Fig. 7. Measured spectrum of the generated analog signal using an USB oscilloscipe (seeFig. 6).

References

[1] L. B. Kish, R. Vajtai, C. G. Granqvist, "Extracting information from the noise spectra of chemical sensors: Electronic nose and tongue by one sensor," Sens. Actuators B, Chem., vol. 71, pp. 55–59, 2000.

[2] L. B. Kish, "Totally Secure Classical Communication Utilizing Johnson (-Like) Noise and Kirchhoff's Law, " Physics Letters A, vol. 352. pp. 178–82, 2006.

[3] G. Marsaglia. "A Current View of Random Number Generators," Keynote Address, Computer Science and Statistics: 16th Symposium on the Interface, Atlanta, 1984

[4] Marsaglia, G. (n.d.). "The Marsaglia Random Number CDROM including the Diehard Battery of Tests," available: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/ [Accessed 26 Apr. 2019].

[5] R. Brown, "Robert G. Brown's General Tools Page," available: https://webhome.phy.duke.edu/~rgb/General/dieharder.php [Accessed 25 Apr. 2019].

[6] R. Mingesz, P. Bara, Z. Gingl, P. Makra, "Digital Signal Processor (DSP) based 1/fα noise generator," Fluctuation and Noise Letters , vol. IV, 2004, L605–L616. p.

[7] NI USB-7856, Multifunction Reconfigurable I/O Device, available: http://www.ni.com/hu-hu/support/model.usb-7856.html [Accessed 26 Apr. 2019].

[8] G. Marsaglia. "Xorshift RNGs," Journal of Statistical Software [Online], vol. 8, Issue 14, July 2003.

[9] Knuth, D. (2014). The art of computer programming, 3rd ed. Upper Saddle River [etc.]: Addison-Wesley, pp.10-26.

[10] M. George, P. Alfke. "Linear Feedback Shift Registers in Virtex Devices," Application Note: Virtex Series and Virtex-II Series, XAPP210 (v1.3) April 2007

[11] M, Matsumoto, T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," ACM Transactions on Modeling and Computer Simulation (TOMACS) - Special issue on uniform random number generation TOMACS Homepage archive, vol. 8, Issue 1, Jan. 1998 , pp. 3-30

[12] National Instruments. "Global Optimization VI", available: http://zone.ni.com/reference/en-XX/help/371361P-01/gmath/global_optimization/ [Accessed 25 Apr. 2019].

[13] Source files, available: http://www.inf.u-szeged.hu/~mingesz/Research/Conference/2019ICNF/ [Accessed 02 May 2019]