

Hardware Acceleration of Parallel Lagged-Fibonacci Pseudorandom Number Generation

Yu Bi, Gregory D. Peterson
Department of Electrical and Computer Engineering
University of Tennessee, Knoxville
{ybi1,gdp}@utk.edu

G. Lee Warren, Robert J. Harrison
Department of Chemistry
University of Tennessee, Knoxville
{gwarren,robert.harrison}@utk.edu

Abstract—The Scalable Parallel Random Number Generators (SPRNG) library is widely used to generate random numbers in Monte Carlo simulations due to the good statistical properties of both its serial and parallel random number streams. In this paper, we suggest an efficient hardware architecture for the Parallel Additive Lagged-Fibonacci Generator (PALFG) provided by the SPRNG library. This design has been implemented on a Virtex-II Pro FPGA device and runs at a clock speed of 125 MHz while delivering one 31-bit random number per clock. Compared to the SPRNG software algorithm executing on a Pentium 4 workstation, a single instance of our design offers a 2.3-fold performance improvement and appears to be 50 times more efficient.

I. INTRODUCTION

Alongside the tremendous advances in serial and parallel computing, the Monte Carlo (MC) method [1] has evolved to tackle many complex problems in such diverse areas as nuclear medicine [2], finance [3], and computational chemistry [4]. Being statistical techniques, MC methods often require a large number of random samples to reduce the statistical error in a computed result to a manageable size. In many cases, the parallel generation of random numbers is employed to meet the intensive number of random samples required.

While fast generators are desirable in many applications, scientific simulations demand good quality random number generators with long periods and minimal correlation. This helps avoid unwanted statistical bias and incomplete sampling. While a significant amount of work has gone into developing good quality generator algorithms, many hardware implementations of random number generators are based on simple Linear Feedback Shift Register (LFSR)-based designs which are largely unsuitable for scientific applications. This is particularly true in the case of parallel applications which may quickly exhaust the supply of random numbers if the generator period is small or may succumb to bias unless the independence of parallel random number streams is rigorously guaranteed. While a variety of good hardware generators such as cellular automata [5], [6], Tausworthe [7], and true noise [8] generators are available, their applicability to high-performance parallel scientific applications is a subject of continuing investigation.

Our approach for hardware-based random number generation is slightly different in that we begin with a software library of proven quality designed specifically for parallel

scientific applications and map that library to hardware. We base our designs on The Scalable Parallel Random Number Generators (SPRNG) library [9] which is widely used for high-performance MC simulations. By porting a commonly used and well characterized software module to hardware, we hope to promote more widespread adoption of hardware acceleration in scientific computing.

The SPRNG library provides a variety of generators including maximal period shift-register, prime modulus linear congruential, multiplicative lagged-Fibonacci, and additive lagged-Fibonacci generators, all of which perform well on the rigorous DIEHARD [10] and NIST [11] test batteries. In this work, we have chosen to implement SPRNG's Parallel Additive Lagged-Fibonacci (PALFG) generator for the production of uniform random integers. This generator is attractive because it is architecturally distinct from common LFSR designs, the bit-width of the output is easily tunable, and it is the most commonly used generator (the default choice) in the SPRNG library. We also note that no hardware implementation of any SPRNG generators has been reported in the literature.

The remainder of this paper is organized as follows. In section II, we discuss the additive lagged-Fibonacci algorithm and the parallel implementation in SPRNG. In section III, we propose a hardware architecture for SPRNG's PALFG generator and discuss various aspects of the design. Resource usage and performance of our implementation is described in section IV.

II. BACKGROUND

The Additive Lagged-Fibonacci Generator (ALFG) [12] is a recurrence-based generator that is parameterized by the values or lags ℓ and k and an initial state array of length ℓ and width m . The transition function

$$x_n = x_{n-\ell} + x_{n-k} \pmod{2^m} \quad (1)$$

describes how a new value x_n is derived from two previous values $x_{n-\ell}$ and x_{n-k} in the sequence. Here, $k < \ell$ and we take m to be equal to 32 for the generation of 32-bit random numbers. Since a new value depends on a prior value at most ℓ positions back in the sequence, it is sufficient to only store the ℓ most recent sequence values in the state array.

The ALFG has two important properties that are particularly advantageous for parallel MC simulations. The first is that

TABLE I
PARAMETERS FOR THE PALFG GENERATORS SUPPLIED BY SPRNG

ℓ	k	m
17	5	32
31	6	32
55	24	32
63	31	32
127	97	32
521	168	32
521	353	32
607	273	32
607	334	32
1279	418	32
1279	861	32

the maximum period [13] of the ALFG is known to be $(2^\ell - 1) \times 2^{m-1}$ or 2×10^{394} for the generator $\{\ell = 1279, k = 861, m = 32\}$. This means that we are unlikely to exhaust the entire sequence when running on the fastest machines of the foreseeable future. Moreover, the ALFG also possesses $2^{(\ell-1)(m-1)}$ independent, full-period sequences. Thus, many distinctly seeded generators (2^{39618} for the $\{1279, 861, 32\}$ generator) can run in parallel without fear of correlation between sequences.

In an attempt to avoid certain correlations present in the pure ALFG algorithm, the SPRNG implementation [14], [15] employs a combination of two independent ALFG sequences, X and Y , to create a new sequence Z . Sequence X is modified by setting the least significant bit to zero and all values in sequence Y are right-shifted by one bit. The two modified sequences are then combined using a bitwise XOR operation. Parallelism is introduced by judicious choice of initial states [16] such that resulting sequences belong to distinct or fully disjoint classes.

SPRNG provides eleven different parameter sets for the PALFG generator (Table I). While all these parameterized generators exhibit acceptable quality, generators with larger values of the lag ℓ demonstrate slightly better quality and are usually preferred in scientific applications. Thus, our proposed design must be able to accommodate the larger parameter sets provided by SPRNG.

At least two hardware implementations [17], [18] of lagged-Fibonacci generators have been previously reported in the literature. Both consider the merits of logic- and RAM-based designs and conclude that RAM-based designs offer the best resource usage and performance with the fewest drawbacks. However, one major difference between these LFGs and the SPRNG PALFG considered here is that the SPRNG ALFGs must advance by two steps for every random number generated. This results in more severe timing constraints than encountered in standard LFG designs. Additionally, we impose a variety of requirements on our design including the sustained throughput of one random number per clock and extensibility to all SPRNG parameter sets, including large lags, that are desirable in scientific applications.

III. HARDWARE DESIGN

Our goal is to implement the SPRNG PALFG generator in FPGA hardware for the purpose of accelerating scientific applications. For high throughput, the design should be able to generate one random number per clock and produce output identical to the SPRNG library. Compactness is also desired but not essential since we do not expect most scientific applications to need more than one generator per hardware device.

To implement the overall design of the SPRNG PALFG generator, two separate ALFG units of identical architecture are required. The output of the first ALFG is right-shifted by one bit by using a shift register. The output of the second ALFG is modified by setting the least significant bit to zero. These modified outputs are then combined in a 32-bit exclusive-or unit. The non-random least significant bit is then stripped from the final output to yield a 31-bit random integer.

The implementation of each ALFG unit appears straightforward, but is complicated by the fact that the SPRNG ALFGs advance by two recurrence steps at each invocation. Since we wish to obtain one random number per clock, we must be careful to avoid some unexpected resource and data hazards.

To illustrate these hazards, consider how SPRNG implements an ALFG having the parameter set $\{17, 5, 32\}$ by examining the required memory reads per iteration (table II). Step 1 and step 2 denote the two recurrence steps that must occur each iteration. The subscripted values X_n refer to values in the sequence. At each iteration, the output of the ALFG is taken from the result of the operation in the column labeled ‘‘Step 1’’. Results for the operation in the second column are only used to advance the state of the generator. After the last iteration listed in the table, the sequence of memory reads repeats.

Resource Hazard: From table II, it is clear that four values must be read and two values must be written back to memory

TABLE II
MEMORY READS PER ITERATION FOR THE $\{17, 5, 32\}$ GENERATOR.

Iteration	Step 1	Step 2
0	X_0 (L0) + X_{12} (K0)	X_1 (L1) + X_{13} (K1)
1	X_2 (L0) + X_{14} (K0)	X_3 (L1) + X_{15} (K1)
2	X_4 (L0) + X_{16} (K0)	X_5 (L1) + X_0 (K0)
3	X_6 (L0) + X_1 (K1)	X_7 (L1) + X_2 (K0)
4	X_8 (L0) + X_3 (K1)	X_9 (L1) + X_4 (K0)
5	X_{10} (L0) + X_5 (K1)	X_{11} (L1) + X_6 (K0)
6	X_{12} (L0) + X_7 (K1)	X_{13} (L1) + X_8 (K0)
7	X_{14} (L0) + X_9 (K1)	X_{15} (L1) + X_{10} (K0)
8	X_{16} (L0) + X_{11} (K1)	X_0 (L0) + X_{12} (K0)
9	X_1 (L1) + X_{13} (K1)	X_2 (L0) + X_{14} (K0)
10	X_3 (L1) + X_{15} (K1)	X_4 (L0) + X_{16} (K0)
11	X_5 (L1) + X_0 (K0)	X_6 (L0) + X_1 (K1)
12	X_7 (L1) + X_2 (K0)	X_8 (L0) + X_3 (K1)
13	X_9 (L1) + X_4 (K0)	X_{10} (L0) + X_5 (K1)
14	X_{11} (L1) + X_6 (K0)	X_{12} (L0) + X_7 (K1)
15	X_{13} (L1) + X_8 (K0)	X_{14} (L0) + X_9 (K1)
16	X_{15} (L1) + X_{10} (K0)	X_{16} (L0) + X_{11} (K1)

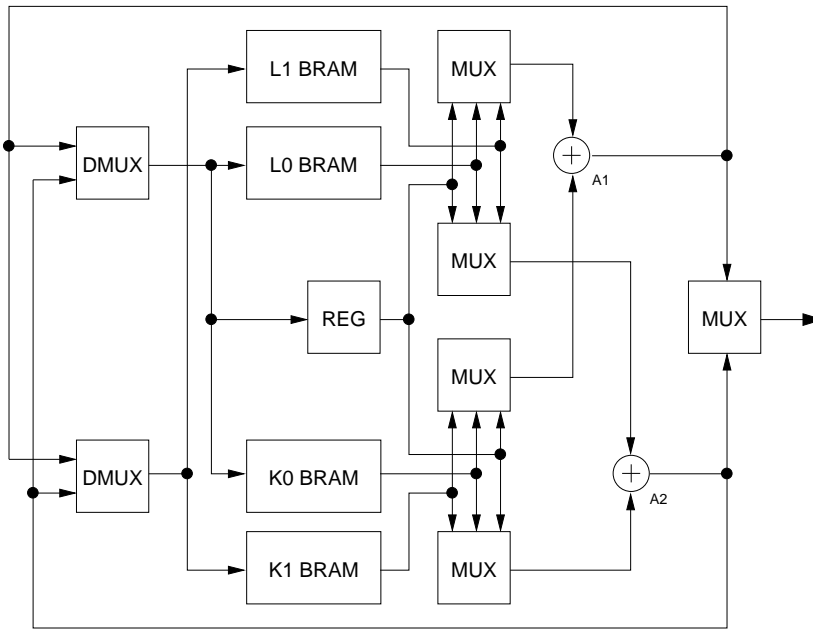


Fig. 1. ALFG architecture and data flow.

in the same clock to sustain throughput. If only dual-port memory resources are used, multiple block RAM (BRAM) resources are required. Our proposed solution requires four BRAM resources labeled L0, L1, K0, and K1. As table II indicates, one value is read from each resource each clock. When even values are updated, they are written back to both the L0 and K0 BRAM resources. Likewise, updates of the odd values are written back to the L1 and K1 memory resources. This ALFG design and data path is illustrated in figure 1.

Resource Hazard: Our above BRAM solution alleviates the above resource hazard, but incidentally creates another. In iteration 8, we see that data must be read from the L0 resource twice in the same clock. Similarly, this happens for the K0 resource in iteration 2. To avoid these two special cases which occur at the modulo ℓ wrap-around, we introduce an auxiliary register, synchronized with the L0 and K0 resources, which is used whenever an extra read from L0 or K0 is required. This eliminates the remaining resource hazard.

Data Hazard: In iteration 2, we must read the value of X_0 which has just been updated in iteration 0. This is potentially the most difficult hazard to resolve since the value must be written back to memory and available for reading within two clocks. Thus, our total data pipeline cannot exceed two stages. Excluding the multiplexer logic in our proposed design, we have exactly two pipeline steps. By using combination logic in the multiplexer units, we can practically achieve two pipeline steps, but the added propagation delay through the multiplexers results in timing hazards that limit the maximum allowable clock speed.

We note that this problem is most acute for the generators with the smallest values of the parameter k as in the $\{17, 5, 32\}$ and the $\{31, 6, 32\}$ generators. For larger lags, where the

pipeline restrictions are relaxed, the multiplexer logic can be converted to registered logic and treated as additional pipeline steps. Thus, this hazard is effectively eliminated by selecting generators with larger lags. This is most convenient for scientific applications where larger lags are usually preferred.

IV. RESOURCE USAGE AND PERFORMANCE

Our ALFG design (figure 1) incorporates four 512×36 bit dual-port block RAMs, one 32-bit register, two 32-bit adders, five multiplexers and two demultiplexers in addition to the necessary counters and controller (not shown). One counter is used for seed downloading when the generator is initialized. Another counter produces memory addresses for the read and write operations. The controller manages data flow through the design by controlling the multiplexer units. More BRAM resources and additional multiplexer logic (not shown) are required for lags larger than 512.

The complete PALFG design incorporates two independent ALFG units, a shift register, and a 32-bit XOR unit and has been targeted for both the Xilinx Virtex-II Pro and Spartan III devices. The design is written in VHDL and has been synthesized using Synplify Pro. Xilinx ISE WebPack 8.1i was used to perform the place and route. The synthesis results and resource usage of our implementation on these devices is shown in table III. For the Virtex-II Pro devices, the output clock rate of 71.4 MHz indicates that our design is able generate about 70 million random numbers per second for the $\{17, 5, 32\}$ generator (the worst case). Parameters with larger lags (employing registered output and additional pipelining) lead to generators capable of 125 MHz or faster clock speeds. In all cases, the output of our hardware PALFG implementation produces results identical to SPRNG.

TABLE III

RESOURCE USAGE AND CLOCK RATE OF THE PRESENT ALFG DESIGNS

FPGA Device	Generator	LUTs	BRAMs	MHz
Spartan3 XC3S200	{17, 5}	1100	4	64.5
Virtex-II Pro XC2VP30	{17, 5}	1100	4	71.4
Virtex-II Pro XC2VP30 ¹	{127, 97}	1100	4	125.0

¹Alternate ALFG design using registered logic and additional pipelining.

The present PALFG design requires more FPGA resources compared with other hardware-implemented uniform random number generators. The primary reason for this is that lagged-Fibonacci generators require a significant amount of memory to store the lagged values. For example, the {1278, 861, 32} generator requires three BRAM resources at minimum. To satisfy the requirement that the SPRNG ALFGs advance twice in one clock, we must double the required BRAM resources.

To evaluate the performance of our design, we have compared it with the SPRNG PALFG software generator running on a Pentium 4 workstation (table IV). We find that the present hardware implementation is a factor of 50 more efficient than the corresponding software generator. Our implementation slightly outperforms the Pentium 4 by a factor of 1.3 (for the {17, 5, 32} generator) or 2.3 (for the {127, 97, 32} generator) even though the clock speed is an order of magnitude slower.

We also note that the relative performance of our hardware implementation can be greatly improved by instantiating several PALFGs on the same FPGA chip. From SPRNG's point of view, these instances are seeded appropriately to provide fully independent, parallel random number streams. On the Virtex-II Pro XC2VP30, we predict that over 12 instances of the PALFG design could be instantiated, providing another factor of 10 gain in efficiency over the Pentium 4.

V. CONCLUSIONS

And efficient hardware implementation of SPRNG's parallel additive lagged-Fibonacci generator has been presented. This fully pipelined uniform random number generator produces one random number per clock at rates of approximately 125 million random numbers per second for most parameter sets

TABLE IV

A COMPARISON OF THE PRESENT HARDWARE IMPLEMENTATION VS SPRNG SOFTWARE RUNNING ON A PENTIUM 4 WORKSTATION

Criteria	Hardware {17, 5, 32}	Hardware {127, 97, 32}	Pentium 4
Clock Speed (MHz)	71.4	125.0	2800
Rate (MRNS) ¹	71.4	125.0	55.6
Efficiency	1.0	1.0	0.02

¹Millions of random numbers per second.

employed by the SPRNG library. At a clock rate of 125 MHz, the present design outperforms a 2.8 GHz Pentium 4 processor by a factor of 2.3. Thus, we believe that the present hardware implementation will benefit scientific applications running on high-performance machines such as the Cray XD1 that employ FPGAs for acceleration of computations.

VI. ACKNOWLEDGMENT

This work was partially supported by the University of Tennessee Computational Science Initiative.

REFERENCES

- [1] N. C. Metropolis and S. M. Ulam, "The Monte-Carlo method," *J. Am. Stat. Assoc.*, vol. 44, p. 335, 1949.
- [2] H. Zaidi and G. Sgouros, Eds., *Therapeutic Applications of Monte Carlo Calculations in Nuclear Medicine*, ser. Series in Medical Physics and Biomedical Engineering. Institute of Physics, 2002, vol. 24.
- [3] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, ser. Stochastic Modelling and Applied Probability. Springer, 2003, vol. 53.
- [4] A. Aspuru-Guzik, O. E. Akramine, J. C. Grossman, and W. A. Lester, "Quantum Monte Carlo for electronic excitations of free-base porphyrin," *J. Chem. Phys.*, vol. 120, p. 3049, 2004.
- [5] P. D. Hortensius, R. D. McLeod, and H. C. Card, "Parallel random number generation for vlsi systems using cellular automata," *IEEE Trans. Comput.*, vol. 38, no. 10, pp. 1466–1473, 1989.
- [6] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider, "High-performance cellular automata random number generators for embedded probabilistic computing systems," in *Evolvable Hardware*. IEEE Computer Society, 2002, pp. 191–200.
- [7] G. Zhang, P. H. W. Leong, D.-U. Lee, J. D. Villasenor, R. C. C. Cheung, and W. Luk, "Ziggurat-based hardware gaussian random number generator," in *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications*. IEEE, 2005, p. 275.
- [8] K. H. Tsoi, K. H. Leung, and P. H. W. Leong, "Compact fpga-based true and pseudo random number generators," in *FCCM*. IEEE Computer Society, 2003, pp. 51–61.
- [9] M. Mascagni, D. Ceperley, and A. Srinivasan, "Algorithm 806: SPRNG: A scalable library for pseudorandom number generation," *ACM Transactions on Mathematical Software*, vol. 26, p. 436, 2000.
- [10] G. Marsaglia, "DIEHARD: A battery of tests of randomness." [Online]. Available: <http://stat.fsu.edu/pub/diehard/>
- [11] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, "A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications," NIST, Tech. Rep. NIST Special Publication 800-22, 2001. [Online]. Available: <http://csrc.nist.gov/rng/>
- [12] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley Publishing Company, 1982, vol. 2.
- [13] S. Aluru, "Parallel additive lagged fibonacci random number generators," in *ICS '96: Proceedings of the 10th international conference on Supercomputing*. ACM, 1996, p. 102.
- [14] I. H. Sloan and H. Wozniakowski, "A fast, high quality, and reproducible parallel lagged-Fibonacci pseudorandom number generator," Supercomputing Research Center, Tech. Rep. SRC-TR-94-115, 1994.
- [15] M. Mascagni, M. Robinson, D. Pryor, and S. Cuccaro, "Parallel pseudorandom number generation using additive lagged-Fibonacci recursions," *Springer-Verlag Lecture Notes in Statistics*, vol. 106, p. 263, 1995.
- [16] D. V. Pryor, S. A. Cuccaro, M. Mascagni, and M. L. Robinson, "Implementation of a portable and reproducible parallel pseudorandom number generator," in *SC*. IEEE Computer Society, 1994, p. 311.
- [17] P. P. Chu and R. E. Jones, "Design techniques of FPGA based random number generator," in *Military and Aerospace Applications of Programmable Devices and Technologies Conference*. The Johns Hopkins University Applied Physics Laboratory, 1999.
- [18] D. B. Thomas and W. Luk, "High quality uniform random number generation for massively parallel simulations in FPGAs," in *International Conference on Reconfigurable Computing and FPGAs*. IEEE Computer Society, 2005, p. 12.