Efficiency and Area Reduction for A PRNG Frame Work Based on Well Method

Jerina Ezhilarasi D[1] Mr. P. Anand Selva Kumar[2]
[1]PG Scholar [2]Assistant Professor
[1]Department of VLSI DESIGN [2]Department of Electronics Communication Engineering
[1,2]MahaBarathi Engineering College, Chinnasalem

Abstract— WELL method mainly describes that the 7.1 fold faster technique is inbuilt to increase the efficiency and reduce the area. The proposed architecture also implemented on targeting different device for the comparison of other types of PRNG. Interleaver sequence technique is used. A large number of research is made on PRNG compared with the MT algorithm specified for less than 423 MHz of frequency range. Now a days the recent communication technique is made for the WELL method along with the PRNG to achieve the high frequency range than the 423MHz. A resource – efficient hardware architecture for WELL with a throughput of one sample per cycle. The Well Equidistributed Long-period Linear (WELL) algorithm is proven to have better characteristics than the Mersenne Twister (MT), one of the most widely used long-period pseudo-random number generators (PRNGs). In this paper, we propose a hardware architecture for efficient implementation of WELL. Our design achieves a throughput of 1 sample-per-cycle and runs as fast as 423MHz on a Xilinx XC6VLX240T FPGA Furthermore, we design a software/hardware framework that is capable of dividing the WELL stream into an arbitrary number of independent parallel sub-streams. With support from software, this framework can obtain speedup roughly proportional to the number of parallel cores. We also apply our framework to a Monte-Carlo simulation for estimating p. Experimental results verify the correctness of our framework as well as the better characteristics of the WELL algorithm.

Key words: WELL, XC6VLX240T FPGA

I. INTRODUCTION

Random numbers are a fundamental tool in many cryptographic applications like key generation, encryption, masking protocols, or for internet gambling. We require generators which are able to produce large amounts of secure random numbers. Simple mathematical generators, like linear feedback shift registers (LFSRs), or hardware generators, like those based on radioactive decay, are not sufficient for these applications. Algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random in that it is completely determined by a relatively small set of initial values, called the PRNG's state. Pseudo-random numbers are important in practice for simulations (e.g., of physical systems using the Monte Carlo method), and are central in the practice of cryptography. Most pseudo-random generator algorithms produce sequences which are uniformly distributed by any of several tests. Common classes of these algorithms are linear Congruential generators, lagged Fibonacci generators, linear feedback shift registers and Generalised feedback shift registers. Recent instances of pseudo-random algorithms include Blum Blum Shub, Fortuna, and the Mersenne Twister. The latter is the subject of this paper study.

The 1997 invention of the Mersenne twister algorithm, by Makoto Matsumoto and Takuji Nishimura [1], avoids many of the problems with earlier random number generators. Indeed, Mersenne twister random numbers have the colossal period of 219937-1 iterations (>43×106000), are proven to be equi distributed in (up to) 623 dimensions (for 32-bit values), and can be generated faster than other statistically reasonable generators. This has made the Mersenne twister increasingly the random number generator of choice for Statistical simulations and generative modeling. Hardware acceleration based on reconfigurable hardware in the form of Field Programmable Gate Array (FPGA) has attracted a great deal of interest in the past 20 years as they offer the high performance of a dedicated hardware solution with the programmability feature. More recently, general purpose comation on Graphic Processing Units (GPUs) has been gaining great popularity as a new field of study (coined GPGPU) which aims at harnessing the parallelism available in GPUs for more general purpose computing applications as opposed to the original purpose these devices have been designed for i.e. graphics processing. With the increasing programmability of commodity GPUs, these relatively cheap and open devices are now being used in scientific computing applications as high performance computing platforms.

High quality random numbers are of critical importance to many scientific applications, particularly for Monte Carlo simulations. Given the advantages of high performance and reproducibility, pseudorandom number generators (PRNGs) based on linear recurrences over F2 are widely adopted in such simulations. One prevalent F2-linear PRNG is the Mersenne Twister (MT), which has very long period and good equidistribution. However, MT is also proved to have certain drawbacks. For example, one serious issue is that it is sensitive to poor initialization and can take a long time to recover from a zero-excess initial state. The well equidistributed long-period linear (WELL) algorithm is proposed to fix this problem. Compared with MT, WELL has better equidistribution while retaining an equal period length As application sizes scale, one emerging trend is to develop parallelized version of the applications to exploit the available parallel hardware resources, such as in field-programmable gate arrays (FPGAs), to achieve high speed in performance. Being the key component of various scientific applications, designing PRNGs that can rapidly provide independent parallel streams of high quality random numbers is also becoming increasingly important in modern systems. The fast jump ahead technique provides an efficient method to determine the starting point of a new sub stream from an existing sub stream, thus allowing multiple PRNGs to generate independent sub streams in parallel and

All rights reserved by www.ijsrd.com

providing strong theoretical support for parallelizing F2-linear PRNGs with long-period. A large body of research is done on F2-linear PRNGs, most of which focus on algorithms and relevant software implementations. Only a few hardware implementations can be found in the literature. For those hardware implementations, most of them employ the MT method, including nonparallel and parallel hardware implementations. With its advantages over MT, WELL also receives great attention from the software community. However, few hardware implementations can be found. The Ukalta Engineering Corporation gave a brief introduction to its product that employs the WELL algorithm. However, it only achieves a throughput of one sample every two cycles and no structural details are revealed.we presented a BRAM-and-register-hybrid architecture for WELL19937 with a throughput of one sample per cycle.

Random number generation is a very important operation in computational science e.g. in Monte Carlo simulations methods. It is also a computationally intensive operation especially for high quality random number generation. In this paper, we present the design and implementation a parallel implementation of one of the most widely used random number generators, namely the Mersenne Twister. The latter is very widely used in high performance computing applications such as financial computing. Implementations of our parallel Mersenne Twister number generator core on Xilinx Virtex4 FPGAs achieve a throughput of 26.13 billion random samples per second. The paper also reports equivalent parallel software implementations running on an Intel Core 2 Quad Q9300 CPU with 8 GB RAM, using multi-threading technology and the Intel® Math Kernel Library (MKL), as well as on an NVIDIA 8800 GTX GPU. Comparative results show that our FPGA-based implementation outperforms equivalent CPU and GPU implementations by ~25x and ~9x respectively. Moreover, when using the same amount of energy, the FPGA can generate 37x and 35x more Mersenne Twister random samples than the CPU and the GPU, respectively. A parallel Mersenne Twister random number generator engine has been presented in this paper. By exploiting deep pipelining, effectively partitioning the assignments for hardware and software, and fully utilizing the FPGA resources on the reconfigurable device, our FPGA implementation outperforms the more mainstream multi-core CPU and GPU platforms by ~25x and ~9x, respectively. Moreover, power measurements showed our FPGA implementation to be 37x and 35x more energy efficient than CPU and GPU respectively. These experiments clearly show the superiority of FPGAs to CPUs and GPUs on speed and energy consumption grounds

The fast generation of random numbers is essential for many tasks. One of the major fields of application are Monte Carlo simulation, for example widely used in the areas of financial mathematics and communication technology. Monte Carlo applications are ideally suited to field programmable gate arrays (FPGAs) because of the highly parallel nature of the applications, and because it is possible to take advantage of hardware features to create very efficient random number generators (RNGs). In particular, uniform random bits are extremely cheap to generate in an FPGA, as large numbers of bits can be generated per cycle at high clock rates using lookup tables, or first-in-first out (FIFO) queues. Many applications are reliant on uniform random numbers, such as monte-carlo integration, simulated annealing, and financial simulations. Such applications require huge amounts of processing power, while offering plenty of scope to exploit fine-grain and coarse-grain parallelism, and so are often ideally suited to implementation in FPGAs.

The availability of high quality Gaussian random numbers is critical to many simulation, graphics and Monte Carlo applications. Currently, the majority of such simulations are performed using systems based on microprocessors ,digital signal processors, or other software-programmable devices. In these systems, the trigonometric, exponential, andother functions involved in many of the methods for obtaining Gaussian random variables can be performed using software libraries [1]. As a result, not much research has been reportedconcerning efficient hardware methods for implementation of Gaussian noise generators. However, well-optimized hardware implementations can often operate one or more orders of magnitude faster than similarly optimized software implementations. Recent advances in field-programmable gate array(FPGA) technology have substantially improved performanceand cost effectiveness of hardware implementations, and they provide a strong motivation for us to reexamine the issue of Gaussian noise generation in hardware. The work described here is originally motivated by ongoing advances in communications relating to channel codes [2], and in particular by the development of new generations of channel codes that operate on very long (thousands to tens of thousands of bits each) blocks of data. For these codes, it is often desirableto perform simulations of extremely large numbers of blocks in order to assess the bit-error rate (BER) performance at rates as low as 10 .There are many other applications in which very large simulations using Gaussian noise are valuable as well.

II. EXISTING SYSTEM

In existing method Random number generators based on LFSR and linear recurrences modulo 2 are among the fastest long-period generators currently available. The uniformity and independence of the points they produce, over their entire period length, can be measured by theoretical figures of merit that are easy to compute, and those having good values for these figures of merit are statistically reliable in general. Some of these generators can also provide disjoint streams and sub streams efficiently. Several widely-used RNGs of this form are not statistically reliable, but some well-designed ones are good, reliable, and fast. The RNGs used for simulation are deterministic algorithms whose output is statistically indistinguishable from a true RNG. They are usually implemented as a recurrence equation, and their output sequence eventually repeats (the length is called the period). A good RNG algorithm must have a long period, an efficient realization and be portable. All arithmetic for these GF(2)-linear RNGs can be performed with elementary bitwise operations, making them highly suitable for hardware implementation.

This focus on G F(2)-linear RNGs is because they have long periods (> 21024 _ 1), well-proven statistical properties and are widely used for software simulations; however, efficient hardware implementations of such long-

All rights reserved by www.ijsrd.com

period RNGs are almost non-existent. To solve this problem and accelerate hardware simulations, we develop parallelized architectures for long-period RNGs and demonstrate specific implementations.

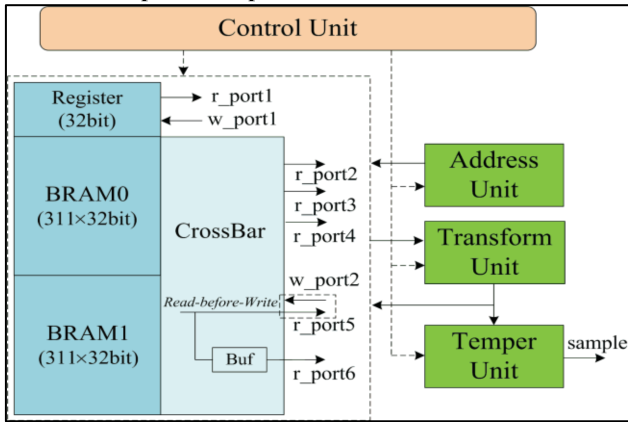

Fig. 1: Control Unit

III. HARDWARE ARCHITECTURE FOR WELL19937

It consists of five blocks: the Control Unit, the Address Unit, the Transform Unit, the Temper Unit, and a 6R/2W RAM. The core component is the RAM, which stores the 624 32-bit state vectors and is capable of concurrently supporting six Reads and two Writes. The Address Unit generates appropriate R/W addresses for the RAM. The Transform Unit and the Temper Unit perform the Transform and Temper operations of the WELL algorithm, and can be fully pipelined. The Control Unit produces the control signals to coordinate the system.

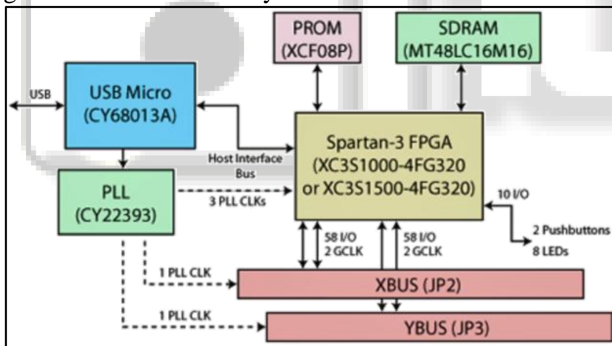

Fig. 2

A. *Structure of the 6R/2W Multiport RAM:*

Based on the transformation process of WELL algorithm, in each generation process, six blocks from the state vector are fetched while two blocks are updated. Therefore, to achieve the expected throughput, the RAM should read six operands and store two results concurrently in a single cycle.

Such a RAM can be directly implemented using 624 32-bit registers, but this is not area-efficient and is impractical when building parallel PRNGs. It is also not straightforward to provide eight ports by simply assembling four BRAMs together, as we need to guarantee that the read and write operations are distributed across different BRAMs evenly. Instead, we propose a BRAM-and-register hybrid structure to build the required 6R/2W multiport RAM, which is the key component to achieve one sample per cycle throughput. The state vector S[0] is read and updated in each cycle. We therefore can use a single register to store S[0] and provide the necessary 1R/1W operations. The BRAMs of the FPGA allow a Read-before-Write operation, i.e., when writing a new data into an address, the data previously stored at this address can be fetched concurrently in the same clock cycle. Using this feature, the w_port2 and the r_port5 can be provided by a single port of a BRAM thus saving one R/W operation. Since the Read addresses of the r_port5 and r_port6 are always adjacent, the data from r_port5 can be buffered and reused by r_port6 in the next clock cycle. This saves one more Read operation. By utilizing these two optimizations, the remaining six R/W operations are decreased to only four operations. This can be provided by two dual-ported 311 × 32-bit BRAMs. Assume the access delay of the BRAM is one clock cycle.

The comparison of a number of FPGA-based PRNGs, in terms of quality metric, resource usage, and performance. The WELL19937, MT19937, LFSR- 160, Tauss-113, and the MLFG and CMFG from HSPRNG are software generators ported to hardware, while the LUT-SR is the one developed specifically for FPGA. For fair comparison, the proposed WELL architecture is implemented targeting different FPGA devices. As expected, the improved WELL generator was more resource-efficient (it saves as much as 50% FPGA resource) compared with the original structure, while achieving a comparable performance. To our knowledge, the only other hardware WELL generator reported is the one introduced in the product brief of the Ukalta Engineering Corporation. This structure is capable of producing one sample every two cycles, which is only half of us. To our knowledge, the fastest FPGA implementations of MT19937 reported. As shown in Table I, although consuming more logic resource (this is because of the algorithm complexity), the WELL generator achieves a bit higher throughput. In addition, it is worth to note that the resource usage of the WELL generator is only about 0.3% of the device, which is negligible. For statistical testing, both WELL and MT fail only two linear complexity tests. However, the characteristic polynomial of MT19937 has only 135 nonzero coefficients out of 19937. In this condition, when the state vector is initialized with a relatively large fraction of zeros, then only a small part of the state will be changed at every generation process and the change to the state will also be very small. This tendency likely continues for many steps and causes the MT generator taking a very long time to recover from zero-excess states. WELL correct this weakness by keeping its characteristic polynomial with the number of nonzero coefficients close to half the degree (e.g., the nonzero number is 8585 out of 19937 for WELL19937)

All rights reserved by www.ijsrd.com

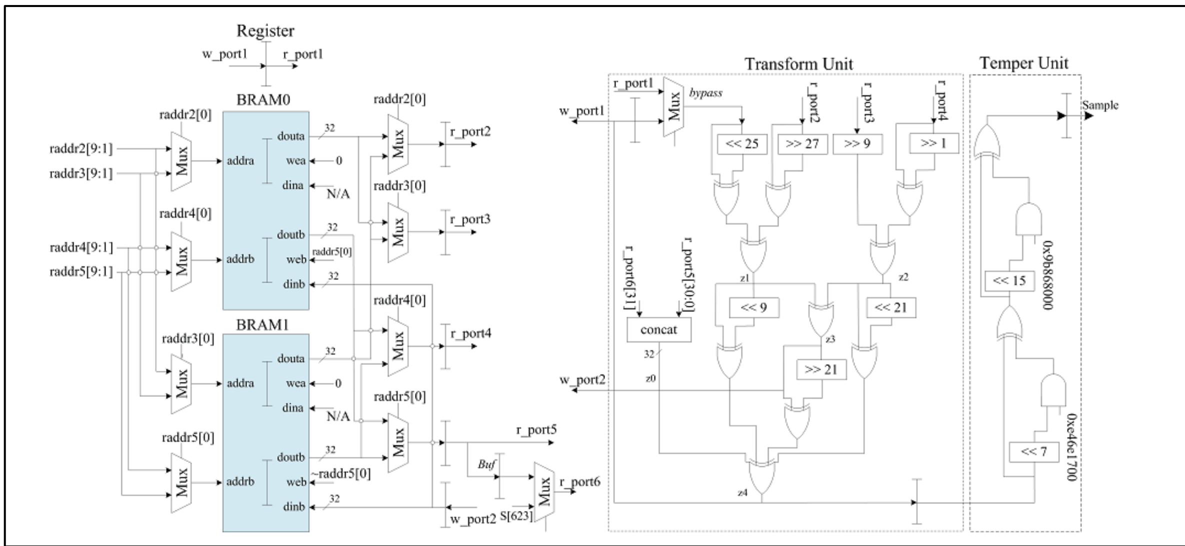

Fig. 3

The LFSR-160 and Tauss-113 behave particularly badly and fail multiple tests, including those do not depend on the linear structure of the generator. The integer generators MLFG and CMRG from HSPRNG (that are based on recurrence modulo a positive integer) pass all tests, but the decimal computations significantly slows them down, thus their performance/cost ratio are not attractive. The LUT-SR- 19937 also fails the two linear tests and is of about the same quality as WELL. Although consuming more resource, the throughput of LUT-SR greatly surpass WELL. This is normal because LUT-SR is optimized specifically for FPGA hence it is intrinsically faster than those hardware-ported generators.

B. *Framework Evaluation:*

The framework is evaluated from two aspects. For the software, we evaluate the average time for the jump process with different steps. Simulation results show that the jump process can be completed within a few milliseconds regardless of the jump distance. For the hardware, we implemented the PRNG array with 1, 4, 8, 12, and 16 simulation cores. Initial states are generated by software and then directly written into each generator

IV. STATISTICAL TESTING

The statistical testing is two-fold: 1) the sequential testing to check for correlations within a stream and 2) the parallel testing to check for correlations between different sub streams. For the sequential testing, we just have verified that the outputs of the hardware and the standard software version are exactly the same when starting with the same seed. Since the statistical properties of the WELL algorithm was already well proven to be good, hence the same should be true of the hardware WELL generator.

The parallel testing is performed by interleaving different substreams of Table II into a single stream. For example, if the parallel degree is n and the stream i is given by xi,0, xi,1, xi,2, . . ., then the new stream is in the form of x0,0, x1,0, . . . , xn−1,0, x0,1, x1,1, . . ., xn−1,1, ... We apply the new stream to the standard statistical test suites, Diehard and Crush from TestU01. Testing results show that the parallel generators pass all tests, which verify the independence of different parallel streams generated by our framework

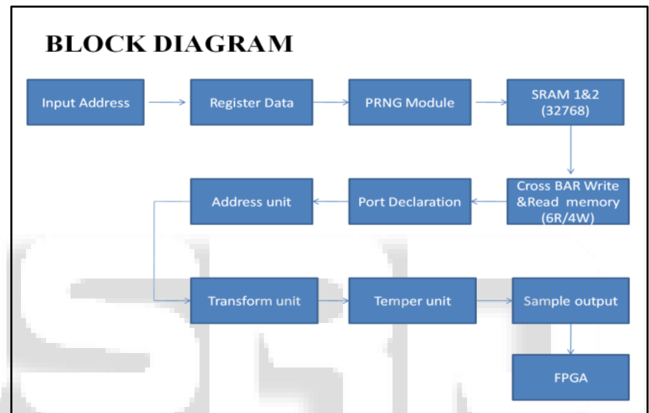

Fig. 4: Block Diagram

V. BLOCK DIAGRAM DESCRIPTION

The input address is given maximum up to 256 bit binary data. Input address mainly requires collecting the input data. The registered data stores the binary data in two modes. One is unimodal and another one is bimodal. It also acts as a buffer. In SRAM, new data is taken from PRNG Module that is spitted in to two units .One is address unit and temper unit. Cross BAR Write and read memory used for editing purpose from the output PRNG module.

It mainly propose the value from the EEPROM and it acts to re - correct and reconfigure the values of the in out signals. The port declaration is used to declare the port configuration of the given input module hence these used to reduce the architecture level and used to maintain the chip level design. It consists of five blocks: the Control Unit, the Address Unit, the Transform Unit, the Temper Unit, and a 6R/4W RAM.

The core component is the RAM, which stores the 128 256-bit state vectors and is capable of concurrently supporting six Reads and two Writes. The Address Unit generates appropriate R/W addresses for the RAM. The Transform Unit and the Temper Unit perform the Transform and Temper operations of the WELL algorithm, and can be fully pipelined. The Control Unit produces the control signals to coordinate the system.

All rights reserved by www.ijsrd.com

An SRAM cell has three different states. It can be in: standby (the circuit is idle), reading (the data has been requested) and writing (updating the contents). The SRAM to operate in read mode and write mode should have "readability" and "write stability" respectively. The three different states work as follows:

A. *PRNG:*

Each PRNG has a finite number of possible states, and hence the “random” sequence will start repeating after a certain “period,” leading to non randomness. Typically, sequences stop behaving like a truly random sequence much before the period is exhausted, since there can be correlations between different parts of the sequence. We shall describe these terms further, and discuss PRNGs and parallel PRNGs (PPRNGs) in greater detail in § 2.1. Many of the RNGs in use today were developed and tested when computational power was a fraction of that available today. With increases in the speed of computers, many more random numbers are now consumed in even ordinary MC computations, and the entire period of many older generators can be consumed in a few seconds. Tests on important applications have revealed defects of RNGs that were not apparent with smaller simulations.

Thus RNGs have to be subjected to much larger tests than before. Parallelism further complicates matters, and we need to verify the absence of correlation among the random numbers produced on different processors in a large, multiprocessor computation. There has, therefore, been much interest over the last decade in testing both parallel and sequential random number generators [8, 9, 4, 10, 11, 12, 13, 14, 15, 11, 16], both theoretically and empirically. While the quality of the PRNG sequence is extremely important, the unfortunate fact is that important aspects of quality are hard to prove mathematically. Though there are theoretical results available in the literature regarding all the popular PRNGs, the ultimate test of PRNG quality is empirical.

Empirical tests fall broadly into two categories (i) statistical tests and (ii) application based tests. Statistical tests compare some statistic obtained using a PRNG with what would have been obtained with truly random independent identically distributed (IID) numbers on the unit interval. If there results are very different, then the PRNG is considered defective. A more precise explanation is given in section 3.1. Statistical tests have an advantage over application-based tests in that they are typically much faster. Hence they permit the testing of a much larger set of random numbers than application-based tests.

Certain statistical tests have become de-facto standards for sequential PRNGs, and PRNGs that “pass” these tests are considered “good.” We wish to note that passing an empirical test does not prove that the PRNG is really good. However, if a PRNG passes several tests, then our confidence in it increases. We shall later describe parallel versions of these standard tests that check for correlations in a PPRNG. It turns out that applications interact with PRNGs in unpredictable ways. Thus, statistical tests and theoretical results are not adequate to demonstrate the quality of a PRNG. For example, the 32-bit linear congruential PRNG CONG, which is much maligned for its well known defects, performed better than shift register sequence R250 in an Ising model application with the Wolff algorithm, though the latter performed better with the Metropolis algorithm.

VI. APPLICATIONS

A. *Monte Carlo Simulation:*

The π Estimation Details of the Monte Carlo implementation and simulation results can be derived from the conference paper. We evaluate the system running on different number of parallel cores ranging from 1 to 15. We observe that: 1) the π values estimated using different number of cores are the same as the software version, which demonstrates that the statistical characteristics do not change after the algorithm is parallelized as well as the correctness of our proposed architecture and framework and 2) WELL shows better equidistribution properties because it produces more accurate results than MT.

B. *GRNG Based on the Box–Muller Method:*

We also apply the WELL framework to construct a framework for producing parallel Gaussian variables, based on the Box–Muller Method. The table polynomial-hybrid method presented in was adopted for the approximation of the elementary functions (sin, square root, and soon). Floating-point operations were converted into fixed-point operations and the word-length optimization model we proposed in was used to maximize the performance/cost efficiency.

We observe that our WELL-based Gaussian design has the best throughput/area ratio and the longest period. We also evaluate the Gaussian framework with 1, 2, 3, 4, and 8 parallel simulation cores, respectively. Simulation results show that: 1) similar to the WELL framework, both the area usage and the throughput scale roughly linear with the number of simulation cores and 2) all the Gaussian samples produced under different simulation cores pass the standard x2 testing, demonstrating that the statistical characteristics of the samples do not change after the algorithm is parallelized, thus further affirming the correctness of our proposed WELL framework. A Field-programmable Gate Array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare). FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, partial re-configuration of the portion of the design and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications.

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—somewhat like many (changeable) logic gates that can be inter-wired in (many) different configurations . Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR.

All rights reserved by www.ijsrd.com

In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

In addition to digital functions, some FPGAs have analog features. The most common analog feature is programmable slew rate and drive strength on each output pin, allowing the engineer to set slow rates on lightly loaded pins that would otherwise ring unacceptably, and to set stronger, faster rates on heavily loaded pins on high-speed channels that would otherwise run too slow. Another relatively common analog feature is differential comparators on input pins designed to be connected to differential signaling channels.

VII. SIMULATION RESULTS

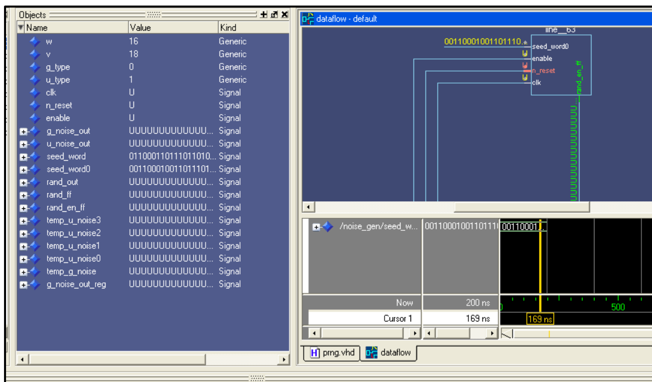

Fig. 5: Simulation Result

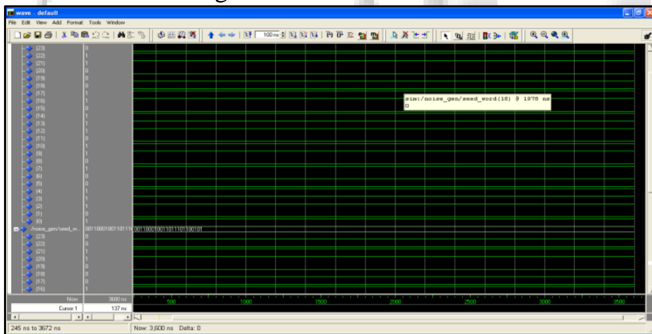

Fig. 6:

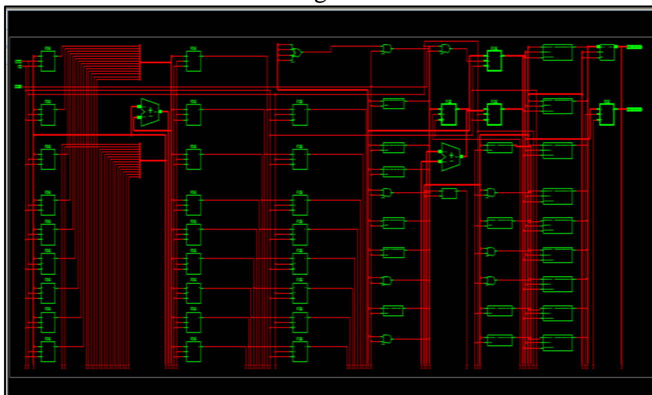

Fig. 7:

VIII. SUMMARY

The parallel testing is performed by interleaving different Sub streams. Testing results show that the parallel generators pass all tests, which verify the independence of different parallel streams generated by our framework. Floating-point operations were converted into fixed-point operations and the word-length optimization model we proposed in was used to maximize the performance/cost efficiency. We observe that our WELL-based Gaussian design has the best throughput/area ratio and the longest period. our proposed one-sample-per-cycle hardware architecture for the WELL algorithm achieved high performance, low area cost, and high quality output at the same time.The well algorithm based random number generator was simulated.

We also evaluate the Gaussian framework with 1, 2, 3, 4, and 8 parallel simulation cores, all the Gaussian samples produced under different simulation cores pass the standard x2 testing, demonstrating that the statistical characteristics of the samples do not change after the algorithm is parallelized, thus further affirming the correctness of our proposed WELL framework. The 7.2 version fold faster technique is used to achieve more than 423MHz. Efficiency is large and area is reduced by using LUT technique. By overcoming the existing system advanced WELL technique is used. Here also 2 samples per cycle is used. Linear PRNG that widely proves that the parallel beams of high quality streams are used. The simulation result is implemented in hardware Spartan 3E kit.

IX. CONCLUSION

Through our study, we demonstrated that our proposed one-sample-per-cycle hardware architecture for the WELL algorithm achieved high performance, low area cost, and high quality output at the same time. The SW/HW framework we develop could parallelize the WELL sequence into arbitrary number of independent parallel sub streams and was successfully applied to two applications. We expect its successful use in various Monte Carlo simulations and other applications.

REFERENCES

[1] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623- dimensionally equidistributed uniform pseudo-random number generator," ACM Trans. Model. Comput. Simul., vol. 8, no. 1, pp. 3–30, Jan. 1998.

[2] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," ACM Trans. Math. Softw., vol. 32, no. 1, pp. 1–16, Mar. 2006.

[3] P. L'Ecuyer and F. Panneton, "Fast random number generators based on linear recurrences modulo 2: Overview and comparison," in Proc. 37th Conf. Winter Simul., 2005, pp. 110–119.

[4] H. Haramoto, M. Matsumoto, T. Nishimura, and P. L'Ecuyer, "Efficient jump ahead for F2-linear random number generators," Inf. J. Comput., vol. 20, no. 3, pp. 385–390, 2008.

[5] V. Sriram and D. Kearney, "An area time efficient field programmable Mersenne Twister uniform random number generator," in Proc. Int. Conf. Eng. Reconfigur. Syst. Algorithms, 2006, pp. 244–246.

[6] C. Shrutisagar and A. Abbes, "High performance FPGA implementation of the mersenne twister," in Proc. 4th IEEE Int. Symp. Electron. Design, Test Appl., Jan. 2008, pp. 482–485.

All rights reserved by *www.ijsrd.com*

[7] S. Konuma and S. Ichikawa, “Design and evaluation of hardware pseudorandom number generator MT19937,” IEICE Trans. Inf. Syst., vol. 88, no. 12, pp. 2876–2879, Dec. 2005.
[8] I. L. Dalal and D. Stefan, “A hardware framework for the fast generation of multiple long-period random number streams,” in Proc. 16th ACM Int. Symp. FPGAs, Feb. 2008, pp. 245–254.
[9] Uncorrelated Pseudo-Random Number Generator IP Cores, Ukalta Engineering Corporation, Edmonton, AB, Canada, 2009.
[10] Y. Li, P. Chow, J. Jiang, and M. Zhang, “Software/hardware framework for generating parallel long-period random numbers using the WELL method,” in Proc. 21st Int. Conf. Field Program. Logic Appl., Sep. 2011, pp. 110–115.
[11] D. B. Thomas and W. Luk, “High quality uniform random number generation through LUT optimised linear recurrences,” in Proc. IEEE Int. Conf. Field-Program. Technol., Dec. 2005, pp. 61–68.
[12] J. Lee, G. D. Peterson, and R. J. Hinde, “Hardware accelerated scalable parallel random number generators for Monte Carlo methods,” in Proc. 51st Midwest Symp. Circuits Syst., 2008, pp. 177–180.
[13] D. B. Thomas and W. Luk, “The LUT-SR family of uniform random number generators for FPGA architectures,” IEEE Trans. Very Large Scale Integrat. (VLSI) Syst., vol. 21, no. 4, pp. 761–770, Apr. 2013.
[14] G. Marsaglia. (1997). DIEHARD: A Battery of Tests of Randomness [Online]. Available: http://stat.fsu.edu/.geo/diehard.html
[15] P. L’Ecuyer and R. Simard, “TestU01: AC library for empirical testing of random number generators,” ACM Trans. Math. Softw. (TOMS), vol. 33, no. 4, p. 22, Aug. 2007.
[16] G. Box and M. Muller, “A note on the generation of random normal deviates,” Ann. Math. Stat., vol. 29, no. 2, pp. 610–611, 1958.
[17] D. Lee, J. Villasenor, W. Luk, and P. Leong, “A hardware Gaussian noise generator using the box-muller method and its error analysis,” IEEE Trans. Comput., vol. 55, no. 6, pp. 659–671, Jun. 2006.
[18] D. Lee, W. Luk, J. Villasenor, and P. Cheung, “A Gaussian noise generator for hardware-based simulations,” IEEE Trans. Comput., vol. 53, no. 12, pp. 1523–1534, Dec. 2004.
[19] G. Zhang, P. Leong, D. Lee, J. Villasenor, R. Cheung, and W. Luk, “Ziggurat-based hardware Gaussian random number generator,” in Proc. Int. Conf. Field Program. Logic Appl., 2005, pp. 275–280.
[20] D. Lee, W. Luk, J. Villasenor, G. Zhang, and P. Leong, “A hardware Gaussian noise generator using the Wallace method,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 13, no. 8, pp. 911–920, Aug. 2005.
[21] I. Paraskevakos and V. Paliouras, “A flexible high-throughput hardware architecture for a Gaussian noise generator,” in Proc. IEEE Int. Conf.Acoust., Speech Signal Process., May 2011, pp. 1673–1676.
[22] E. Boutillon, Y. Tang, C. Marchand, and P. Bomel, “Hardware discrete channel emulator,” in Proc. Int. Conf. High Perform. Comput. Simul., 2010, pp. 452–458. [23] Y. Li, P. Chow, J. Jiang, M. Zhang, and S. Wei, “Software/hardware framework for generating parallel Gaussian random numbers based on the Monty Python method,” in Proc. Int. Conf. Field-Program. Technol., 2012, pp. 190–197.

All rights reserved by *www.ijsrd.com*