FPGA Based Design of Parallel Long-Period Random Number Generation Framework Using WELL Method

Authors
Vanga Rajender¹, D.Srinivas², Pual Chow³
Depart. of Electronics & communication Engineering Vaagdevi College of Engineering, Bollikunta, Warangal, Telangana, India.
Email: rajender431v@gmail.com, cnu0212@gmail.com.

ABSTRACT:
This paper presents a hardware architecture for efficient implementation of the well equi distributed long-period linear (WELL) algorithm. Our design achieves a throughput of one sample-per-cycle and runs as fast as 423 MHz on a Xilinx XC5VFX130T field programmable gate array (FPGA) device. This performance is 7.1-fold faster than a dedicated software implementation. The proposed architecture is also implemented on targeting different devices for the comparison of other types of pseudorandom number generators. In addition, 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. The sequences produced by the single design are verified to be consistent with the standard software generator. In addition, the statistical tests of interleaved sequences are also performed to check for correlations between different sub-streams of the parallel framework. We apply our framework to two applications. Experimental results verify the correctness of our framework as well as the better characteristics of the WELL algorithm compared with the Mersenne Twister method.

I. INTRODUCTION
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 equi-distribution.

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 longtime to recover from a zero-excess initial state. The well equi-distributed long-period linear (WELL) algorithm is proposed to fix this problem. Compared with MT,
WELL has better equi distribution 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 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. In, 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. In, we presented a BRAM-and-register-hybrid architecture for WELL19937 with a throughput of one sample per cycle. This paper is a further work of the conference paper. We propose a more resource-efficient structure that reduces the usage of BRAMs from four to two, while retaining the same throughput. The total resource used is also reduced as much as 50% compared with the original structure. We also design a software/hardware framework to parallelize its output stream based on the new structure. More specifically, we make the following contributions.

1) A resource-efficient hardware architecture for WELL with a throughput of one sample per cycle.
2) A dedicated 6R/2W RAM structure for WELL, which is capable of providing six Reads and two Writes concurrently in a single cycle, with little resource overhead.
3) A software/hardware framework to generate parallel random numbers.
Compared with the conference paper, we make the following improvements:

1) Section II is the main contribution of this paper and is totally rewritten to describe the improved TOWBRAMs-based WELL architecture.

2) In Section IV, the implementation of the WELL structure is discussed more detailed and the comparison is enhanced by introducing more types of FPGA-based PRNGs. The evaluation results are also analyzed in more depth.

3) In Section V, the statistical testing is enhanced by adding parallel testing to check for correlations between different sub-streams.

4) In Section VI, the verification of the WELL framework is strengthened by applying it to a completely new application: Parallel GRNG based on the Box–Muller method.

II. HARDWARE ARCHITECTURE FOR WELL19937

Fig. 1 shows our 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.

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.

```plaintext
// addr[2..4]: access addresses for r_port[2..4].
// addr[5]: access addresses for r_port5 and w_port2.
// pl_tu: the number of pipeline in the Transform Unit.
1. let addr[2..5] = [69,178,448,621]
2. for each clock cycle do
3.   if in the first (pl_tu + 1) clock cycles do
4.     output addr[2..5] as the addresses for r_port[2..5].
5.   else
6.     output addr[2..5] as the addresses for r_port[2..5].
7.     output addr[5] as the address for w_port2.
8.   endif
9.   for j = 2 to 5 do
10.  if addr[j] equal to 0 then
11.    addr[j] = 621 - (pl_tu + 1);
12.  else
13.    addr[j] = addr[j] - 1;
14.  endif
15.  endfor
16. endfor
```

Fig. 2. Pseudocode for generating the access addresses implemented by the Address Unit. Assume the access delay of BRAM is one clock 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 $r_{port5}$ 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 $311 \times 32$-bit BRAMs. Assume the access delay of the BRAM is one clock cycle, Fig.2 shows the pseudo code for generating the access addresses for the Read ports from $r_{port2}$ to $r_{port5}$ (addr[2..5]) and the Write port $w_{port2}$ (addr[5]). The crossbar implements the mapping rules shown in Fig. 3, to forward the R/W ports to appropriate BRAMs, where the access address is generated by the Address Unit based on the pseudo code in Fig.2.

Assume the Transform Unit is divided into a 1-stage pipeline (i.e. $pl_{tu}$ = 1 in Fig. 2). Based on the mapping rules, in the first three clock cycles, the addresses of the R/W ports (as shown in line 1 of Fig. 2) are mapped into BRAM[1, 0, 0, 1], BRAM[0, 1, 1, 0], and BRAM [1, 0, 0, 1], respectively. During runtime, each of

\[
\begin{align*}
\text{BRAM ID} & = \text{access address mod 2} \\
\text{internal BRAM address} & = \frac{\text{access address}}{2}
\end{align*}
\]

Fig. 3. Address mapping rules implemented by the crossbar.

Fig. 4. R/W details during the first four clock cycle.
these addresses is updated synchronously by the same counter and traverses the BRAMs in exactly the same manner. The write address for w_port2 is always the same as the Read address of r_port5 and can be accessed concurrently through the Read before-Write operation. Therefore, no BRAM will need more than two ports in a single cycle. Fig. 4 shows the R/W details during the first four clock cycles, where FB1 and FB2 are the two results generated in the transform step. The numbers in the cells (178, 448, 69, 621, and so on) correspond to the access addresses generated by the Address Unit, which are mapped to the appropriate BRAMs. The R/W accesses to a same address are performed using the Read-before-Write operation and a buffer is used to cache the data shared by r_port5 and r_port6. We can see that 6R/2W operations can be finished in a single cycle. Thus, our BRAM-and-register-hybrid structure can successfully satisfy the required memory accesses.

III. SOFTWARE/HARDWARE FRAMEWORK

Fig. 5 shows the structure of the framework. The jump ahead unit in Software is responsible for:

1) generating the initial vector states for each PRNG according to user configurations, i.e., the total random numbers, the number of simulation cores and the initial seed
2) offloading initial state vectors to the hardware; and
3) dealing with the simulation results.

The core component of the hardware is a PRNG Array consisting of a number of parallel WELL PRNGs. It can be constructed by simply replicating the single generator described in Section II. After receiving proper state vectors, an array of size \( n \) can produce \( n \) random numbers in parallel in every clock cycle, which are denoted as \( y_0, y_1, \ldots, y_{n-1} \).

IV. IMPLEMENTATIONS AND EVALUATIONS

A. Single WELL Generator

Fig. 6 shows the detailed hardware implementation of the WELL architecture. The crossbar is implemented using eight multiplexers. Based on the address mapping rule shown in Fig. 3, the least significant bit of the access addresses are used as the selector while the other nine bits are used as the internal addresses. The embedded output registers of both BRAMs are enabled to improve the clock speed. As one result of the previous iteration is a source operand for the next generation process, this value is directly passed to the next iteration. The verilog source code of the implementation is available at http://sdrv.ms/VomTfS.
Table I shows 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 we presented in, 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 in the literature was the one presented in. 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). In addition to this, WELL also has better equi distribution compared with MT (this point was detailed described in [2] and [13]).
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. The resources usage and performance are shown in Table II. We can see that the throughput/area efficiency roughly remains constant as the number of simulation cores increases.

V. 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
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 sub-streams of Table II into a single stream. For example, if the parallel degree is $n$ and the stream is given by $x_0, 0, x_1, 1, x_2, 2, \ldots$, then the new stream is in the form of $x_0, 0, x_1, 0, \ldots, x_n-1, 0, x_0, 1, x_1, 1, \ldots, x_n-1, 1, \ldots$. We apply the new stream to the standard statistical test suites, Diehard and Crush from TestU01 [15]. Testing results show that the parallel generators pass all tests, which verify the independence of different parallel streams generated by our framework.

### VI. APPLICATIONS

#### A. Monte Carlo Simulation: The $\pi$ 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 $\pi$ 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 equi-distribution 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 so on). 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. Table III shows comparisons of our design against other FPGA-based Gaussian generators. 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 $x^2$ 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.

### VII. CONCLUSION

Through our study, we demonstrated that our propose done-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