A Hardware Framework for the Fast Generation of Multiple Long-period Random Number Streams

Ishaan L. Dalal and Deian Stefan
S*ProCom^2 // Dept. of Electrical Engineering
The Cooper Union, New York, NY 10003
{ishaan, stefan}@cooper.edu

ABSTRACT

Stochastic simulations and other scientific applications that depend on random numbers are increasingly implemented in a parallelized manner in programmable logic. High-quality pseudo-random number generators (PRNG), such as the Mersenne Twister, are often based on binary linear recurrences and have extremely long periods (more than 2^{1024}). Many software implementations of such PRNGs exist, but hardware implementations are rare. We have developed an optimized, resource-efficient parallel framework for this class of random number generators that exploits the underlying algorithm as well as FPGA-specific architectural features. The framework also incorporates fast "jump-ahead" capability for these PRNGs, allowing simultaneous, independent sub-streams to be generated in parallel by partitioning one long-period pseudo-random sequence.

We demonstrate parallelized implementations of three types of PRNGs – the 32-, 64- and 128-bit SIMD Mersenne Twister – on Xilinx Virtex-II Pro FPGAs. Their area/throughput performance is impressive: for example, compared clock-for-clock with a previous FPGA implementation, a "two-parallelized" 32-bit Mersenne Twister uses 41% fewer resources. It can also scale to 350 MHz for a throughput of 22.4 Gbps, which is 5.5x faster than the older FPGA implementation and 7.1x faster than a dedicated software implementation. The quality of generated random numbers is verified with the standard statistical test batteries *diehard* and *TestU01*. We also present two real-world application studies with multiple RNG streams: the Ziggurat method for generating normal random variables and a Monte Carlo photon-transport simulation.

The availability of fast long-period random number generators with multiple streams accelerates hardware-based scientific simulations and allows them to scale to greater complexities.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware; Gate Arrays*; G.3 [**Probability and Statistics**]: Random Number Generation

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Random Number Generator, Mersenne Twister, FPGA, Parallelized Architecture

1. INTRODUCTION

Random numbers are at the core of a wide variety of scientific and computational applications, such as the simulation of stochastic processes, numerical integration by random sampling (Monte Carlo) and cryptography. Since these applications can often be parallelized in a coarse or fine-grained fashion, they have increasingly been implemented on FPGAs in recent years. FPGA-based simulations exist for fields as diverse as cellular biochemical interactions [25], the modeling of interest rates on stock derivatives and radiation/photon transport [22].

The quality of the random number generator (RNG) used for an application can critically affect the accuracy of its results/outputs; there are numerous examples of 'high-quality RNGs' that were found to be statistically 'bad' [1, 2, 5] and led to 'systematically inaccurate' results [26] in simulations using them. Thus, as applications are parallelized on to FPGAs, RNGs that can provide fast streams of statistically reliable random numbers are required.

In this paper, we develop parallelized architectures for the long-period class of RNGs that have been well-proven for use in software applications. Also presented are parallelized hardware implementations of three variants of the *Mersenne Twister* [18] long-period RNG family, as well as a 'jump-ahead' technique that allows partitioning of one long RNG output sequence into multiple independent sub-sequences. Two sample applications (a Monte Carlo simulation and the Ziggurat algorithm for generating normal random variates) complete the hardware framework.

1.1 Types of Random Number Generators

Random number generators (RNG) fall into two broad classes: true RNGs and pseudorandom RNGs.

True Random Number Generators (TRNGs) implemented in hardware are based on an underlying random physical process, such as sampling a fast clock by a slow trigger from amplified thermal noise in a resistor [4]. TRNGs usually need special hardware, are too slow for computational applications (on the order of tens to hundreds of kilobits/sec) and cannot be made to repeat a specific output sequence.

Computational applications almost always use *pseudorandom* number generators (PRNGs). PRNGs are deterministic algorithms, producing output that *behaves* statistically like a sequence of independent random numbers. PRNGs are implemented as a recurrence, and their output sequence will eventually repeat. The length of the sequence is called the *period* of the PRNG. PRNGs have an internal state that must be initialized with a 'seed'; all outputs (and future states) are derived from this initial state. PRNG sequences are therefore repeatable if initialized with the same seed.

An example of a classical PRNG still sometimes used in practice [5] is the *Multiplicative Linear Congruential Generator*, whose state at step n is the integer x_n, with the recurrence:

$$x_n = (a x_{n-1}) \bmod m, \tag{1}$$

where m and a are positive integers.

Note: We shall refer to PRNGs as simply RNGs from this point.

1.2 Requirements for 'Good' RNGs

Random number requirements for simulation and cryptography applications differ. Simulations usually require random variables from specific distributions (normal, exponential, etc.) that are produced by transforming uniformly distributed independent random numbers. A 'good' RNG must also have a long period that exceeds the maximum possible number of random inputs a simulation would need over its expected run-time. Statistical analysis suggests that, to err on the safe side, the period should be the square or even the cube of that maximum [10].

The RNG must be efficiently implementable, be repeatable (for debugging or re-running a simulation) and be portable across different hardware/software platforms. Additionally, the paramount criterion for cryptographic use is *unpredictability*; no computationally feasible algorithm should be able to predict future values of the output sequence based on knowledge of its previous values. Unpredictability is different from simply being sufficiently random based on statistical analysis. For example, if an adversary has access to the output of a linear RNG, they can discover the recurrence defining the PRNG (by using the Berlekamp-Massey algorithm [16]) and predict or generate future output; this would be disastrous for a cryptographical application.

For applications that need multiple input variables as well as simulations that implement *variance reduction* [8] (explained in section 2.2), RNGs with the above-mentioned properties that also provide multiple independent streams are desirable.

1.3 Outline

We are interested in RNGs whose recurrence is linear and defined in binary, i.e. over the finite (or *Galois*) field GF(2) with elements $\{0, 1\}$ (as opposed to the integer-based recurrence of (1)). All arithmetic for these GF(2)-*linear* RNGs is performed modulo 2 and can be implemented very efficiently in hardware using elementary bit-wise operations such as exclusive-ORs (XORs), bit-masks and shifts.

In particular, we focus on GF(2)-linear RNGs with extremely long periods ($\geq 2^{19937} - 1$, although we define 'long' as $\geq 2^{1024}$) and proven statistical properties. The output stream of these long-period RNGs can be partitioned to produce multiple independent sub-streams. Using a recently proposed 'jump-ahead' technique, the starting point of a new sub-stream can be quickly determined from an existing sub-stream; this would allow multiple RNGs to simultaneously generate independent sub-streams. The most common long-period PRNG is probably the *Mersenne Twister* [18]. Despite being extremely popular for software simulation, very few optimized hardware implementations exist in the literature.

This paper is organized as:

- **Background and Related Work:** Brief mathematical preliminaries and the need for multiple streams. Related efforts at implementing long-period GF(2)-linear PRNGs in hardware, what differentiates our contribution and how it advances the state of the art.

- **Mathematical Structure:** The matrix-recurrence structure of the PRNGs as well as the practical word-wise implementation of such recurrences. The mathematical concepts behind 'jumping ahead' in an output sequence.

- **Framework for Parallelized Implementation:** Two methodologies for parallelized implementations of long-period GF(2)-linear PRNGs on FPGAs, including the degree of parallelization possible, how best to exploit FPGA architectural features and the speed/resource usage tradeoffs involved. Integrating multiple-stream RNGs and applications using a soft microprocessor for control and jump-ahead.

- **Implementations:** FPGA implementations of three types of long-period RNGs in various parallelized configurations on the Virtex-II Pro FPGA using the principles of the framework. Performance benchmarks and analysis. To our knowledge, only one of these RNGs (32-bit Mersenne Twister) has previously been implemented in hardware.

- **Applications:** Hardware implementations of two representative applications integrated with multiple RNG streams on an FPGA: a uniform → normal distribution transform (Ziggurat) and a Monte-Carlo simulation of heat-transfer/photon-transport.

- **Statistical Testing and Conclusion:** Verify that the outputs of RNGs designed with our framework pass standard statistical test batteries. Observations and conclusions.

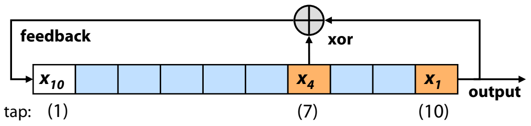


Figure 1: Shift-register LFSR for polynomial $z^{10} + z^7 + 1$

2. BACKGROUND AND RELATED WORK

Let $x = \{x_n\}$ be the binary sequence generated by the linear recurrence over GF(2)

$$x_n = a_1 x_{n-1} \oplus a_2 x_{n-2} \oplus \cdots \oplus a_k x_{n-k}, \tag{2}$$

where $a_i (i = 1, 2, \ldots, k)$ are binary coefficients and $\oplus$ is the XOR. The maximal period of this sequence is $2^k - 1$ and is achieved only if its characteristic polynomial

$$P(z) = z^k + a_1 z^{k-1} + \cdots + a_{k-1} z + a_k \tag{3}$$

is primitive over GF(2)[5, 28]; such a recurrence is called a maximal recurrence.

A classic example of a GF(2)-linear PRNG is the well-known Linear Feedback Shift-Register (LFSR) [28]. LFSRs directly implement maximal recurrences in the form of (3), with a shift register holding the k-bit *state vector*. Each clock cycle, the state is shifted to output one bit and the state bits corresponding to the non-zero coefficients (or 'taps') of the polynomial are XOR'd and fed back into the input. Fig. 1 shows a 10-bit maximal LFSR with period $2^{10} - 1$ and characteristic polynomial $z^{10} + z^7 + 1$.

Bit-output LFSRs with period $2^k - 1$ require k flip-flops and become less area-efficient as their period (and the number of taps) increases, especially for applications that need random *words*. Using word-wise (instead of bit-wise) recurrences led to Generalized Feedback Shift Registers (GFSR) [12]. If $\mathbf{x}$ is the state vector (array) for a GFSR and M_i are integer offsets, the linear recurrence for word $\mathbf{x}[j]$ is

$$\mathbf{x}[j] \leftarrow \mathbf{x}[j + M_1] \oplus \mathbf{x}[j + M_2] \tag{4}$$

GFSRs have certain drawbacks that were corrected by adding a "twist" matrix (A) to the above recurrence,

$$\mathbf{x}[j] \leftarrow \mathbf{x}[j + M_1] \oplus (\mathbf{x}[j + M_2])A \tag{5}$$

resulting in the *Twisted* GFSR (TGFSR) [17]. All the long-period RNGs we use in this paper are derived from TGFSRs.

The representative for our class of long-period RNGs is the *Mersenne Twister* (`MT19937`) [18], a TGFSR variant with 32-bit outputs, a period of $2^{19937} - 1$ and excellent statistical properties. `MT19937` has become very popular for software simulations; for example, it is the default RNG for Matlab, Maple, the *R* statistical language and the GNU Scientific Library. Source codes for `MT19937` are available in a variety of languages.

2.1 Previous and Related Implementations

Despite its popularity, hardware implementations of `MT19937` are rare in the literature [6, 7, 22, 27]. Most of these are straightforward unparallelized implementations of the original C-code, with the exception of [6]. [6] includes a specific example of one of our parallelization methodologies, but their implementation is unoptimized and would require pseudo quad-ported block-memories with data duplication to perform as described (contemporary FPGA block RAMs are dual-ported at best).

Related work on GF(2)-linear PRNGs includes LFSRs designed with recurrences that optimize logic-usage (LUTs) on FPGAs [29] as well as combining multiple LFSR streams to increase the overall period. These are *not* comparable to the long-period RNGs because of their relatively short periods ($< 2^{258}$); this is also true for other RNGs such as `KISS` (period 2^{127}). We also do not consider hardware implementations of non-linear PRNGs such as cellular automata, or of linear PRNGs with recurrences over other finite fields or the integers.

Since the original 32-bit Mersenne Twister, new variations including a 64-bit version (`MT19937-64`) [20] and a 128-bit SIMD (Single Instruction Multiple Data)-oriented Fast Mersenne Twister (`SFMT`) RNGs have been published. Recently, a technique for fast jump-ahead in the output sequences of these long-period RNGs has also been proposed [3]. To our knowledge, no hardware implementations for these long-period RNGs or jump-ahead exist in the literature.

2.2 'Jumping Ahead' and the Need for Multiple Streams

Often, multiple random number inputs to an application are supplied by running parallel copies of a PRNG that have been seeded randomly. The implicit assumption that this produces uncorrelated and non-overlapping sub-sequences of the original PRNG sequence is not supported theoretically. In fact, inter-sequence correlations have been found that affect simulation results in such situations [26].

A valid method of producing parallel sub-sequences is by 'jumping' copies of the basic long-period PRNG ahead in their output sequence by an appropriate amount, which guarantees that the sub-sequences will not overlap. Such a jump-ahead must be efficient, i.e. it should not be necessary to go through an entire sub-sequence to find the starting point for the next sub-sequence.

Multiple streams are also essential for simulations implementing *variance reduction techniques* [8]. Essentially, simulating with random inputs produces random outputs, and data is interpreted by statistically analyzing outputs of simulations run a large number of times and/or multiple simulations with varying parameters. If the *variance* of the simulation output can be reduced without affecting their *expectation* (mean), a smaller number of simulations or shorter run-times may suffice leading to cost and time savings.

3. MATHEMATICAL STRUCTURE OF GF(2)-LINEAR RNGS

3.1 Matrix Recurrence

Following the notation of [9], consider an RNG defined by a matrix linear recurrence over the finite field GF(2):

$$\mathbf{x}_n = \mathbf{A}\mathbf{x}_{n-1}, \tag{6}$$

$$\mathbf{y}_n = \mathbf{B}\mathbf{x}_n, \tag{7}$$

where $\mathbf{x}_n$ is the k-bit state vector at step n, $\mathbf{y}_n$ is the w-bit output vector at step n (k and w are positive integers), $\mathbf{A}$ is a $k \times k$ *transition* (or recurrence) matrix and $\mathbf{B}$ is a $w \times k$ *output transformation* (or 'tempering') matrix, both with elements in GF(2). The characteristic polynomial of the recurrence matrix $\mathbf{A}$ is

$$P(z) = \det(z\mathbf{I} + \mathbf{A}) = z^k + a_1 z^{k-1} + \cdots + a_{k-1}z + a_k, \tag{8}$$

where $\mathbf{I}$ is the identity matrix and each $a_j \in$ GF(2). $P(z)$ must be primitive if the RNG has a maximal period of $2^k - 1$.

For long-period RNGs, k is usually chosen to be a *Mersenne prime*[1] number to make testing the characteristic polynomial for primitivity computationally feasible [19, 5]. The recurrence matrix $\mathbf{A}$ is sparsely structured so that the new state $\mathbf{x}_n$ in (6) can be calculated with some extra word-wise binary operations on $\mathbf{x}_{n-1}$. The output transformation matrix $\mathbf{B}$ is optional, i.e. it can simply be a $w \times w$ identity matrix followed by $w - k$ columns of zeros. However, when better equidistribution (i.e., 'randomness') of the lower-order bits (LSBs) of the output is desired, $\mathbf{B}$ can be a *Matsumoto-Kurita* [17] 'tempering' matrix that applies additional binary operations to the output of (6).

3.2 Practical Word-wise Recurrences

Practical implementations of long-period RNGs partition the state vector $\mathbf{x}$ into N w-bit words such that $k = Nw - r$ ($0 \leq r < w$). The state vector is then updated one word $\mathbf{x}[j]$ at a time ($0 \leq j < N$), where $\mathbf{x}[j]$ is computed in general by binary operations on one or more *near* recurrences (i.e. words that immediately precede or follow $\mathbf{x}[j]$ in $\mathbf{x}$, excluding $\mathbf{x}[j]$ itself) and one or more *far* recurrences $\mathbf{x}[j + M_i]$, where the M_i are positive integer constants. It is easiest to illustrate this with an example. Consider Fig. 2, which shows a twisted GFSR with one near and one far recurrence. The recurrence for the current word $\mathbf{x}[j]$ is defined as

$$\mathbf{x}[j] \leftarrow (\mathbf{x}[j]^{w-r} \,||\, \mathbf{x}[j+1]^r)A \oplus \mathbf{x}[j + M], \tag{9}$$

where $w-r$ upper bits of $\mathbf{x}[j]$ are concatenated ($||$) with r lower bits of the near recurrence word $\mathbf{x}[j+1]$ into a w-bit word, upon which bit-wise operations corresponding to a $w \times w$-matrix A (*not* the state recurrence matrix $\mathbf{A}$) are performed. The result is then XOR'd with the far recurrence word ($\mathbf{x}[j + M]$) to form the updated $\mathbf{x}[j]$;

[1]Mersenne primes are prime numbers that are also one less than a power of two; $2^{19937} - 1$ is the 24th Mersenne prime.

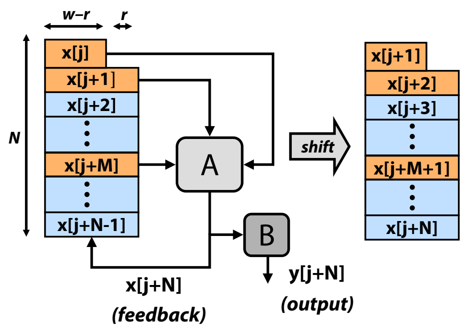


Figure 2: State Transition in a Generic GFSR (block $\boxed{\mathbf{A}}$ implements the recurrence of (9); block $\boxed{\mathbf{B}}$ the optional 'tempering' of section 3)

this is fed back into the shift register and the state vector shifts by one word as shown in Fig. 2. Note that the number of upper/lower bits in (9) are chosen so that actual length of the state vector $k = Nw - r$ is used instead of its effective length Nw. For $r = 31$ and $M = 397$, (9) is the recurrence for the 32-bit Mersenne Twister `MT19937` with period $2^{19937} - 1$, i.e. $w = 32$, $k = 19937$ and $N = 624$.

N w-bit registers can be used to directly implement the recurrence, but this is not area-efficient. For `MT19937`, such an implementation [6] required 20,101 Altera logic elements. Instead, RAM is generally used to store the state vector and the addresses for the current word and its recurrences 'move' (modulo N) at each step rather than the contents of the state vector.

3.3 Fast Jump-Ahead

Suppose we want to jump a long-period RNG with a k-bit state vector, current state $\mathbf{x}_n$ and a matrix recurrence (6), by ν steps to a new state $\mathbf{x}_{n+\nu}$ (ν is large, e.g. $\nu > 2^{100}$). Naïvely, $\mathbf{A}^\nu$ can be pre-computed (mod 2) and then

$$\mathbf{x}_{n+\nu} = \mathbf{A}^\nu \mathbf{x}_n \tag{10}$$

Using standard square-and-multiply exponentiation [5] takes $O(k^3 \log \nu)$ operations and requires k^2 bits for storing $\mathbf{A}^\nu$. For large k such as with `MT19937`, the exponentiation is slow and the 19937×19937-bit $\mathbf{A}^\nu$ needs ≈ 47.4 megabytes of memory!

Recently, a faster approach has been proposed [3] based on the characteristic polynomial (8) of $\mathbf{A}$. The mathematical mechanics are detailed in Appendix A. An auxiliary polynomial $g(z)$ that depends on the size of the jump-ahead ν and $p(\mathbf{A})$ is first computed; $g(z)$ has at most $k - 1$ non-zero binary coefficients. Then, by running the RNG through at most $k - 1$ states, the new state $\mathbf{x}_{n+\nu}$ can be calculated as the sum (XOR) of the states corresponding to the non-zero coefficients of $g(z)$.

$g(z)$ must be calculated once for the type of RNG and is independent of the actual value of the current state $\mathbf{x}_n$. The initial states for a bank of b RNGs that differ by a constant ν can be calculated by iteratively jumping one RNG with state $\mathbf{x}_n$ ahead as

$$\mathbf{x}_{n+(b+1)\nu} = \mathbf{A}^\nu \mathbf{x}_{n+b\nu}$$

The $g(z)$ is calculated on a computer; the actual implementation of the jump-ahead is discussed in the next section.

4. THE HARDWARE FRAMEWORK

We now discuss the architectural optimizations underlying our two methodologies for parallelizing long-period RNGs, the methodologies themselves, their trade-offs and the structure of the overall framework.

4.1 General Optimizations

- *Buffering Near Recurrences:* To save reads, near recurrences can be buffered for re-use.

- *Dual-port Memories/Read-before-Write:* Block RAMs on contemporary FPGAs are usually dual-ported, supporting two asynchronous operations to independent addresses per cycle. Based on the recurrence for a particular RNG, both ports can be used to reduce overall block RAM usage.

  Additionally, block RAMs may allow 'Read-before-Write' operation, where the old data at an address is output while simultaneously writing new data to that address. This feature can almost universally save one read access per cycle by writing the updated (feedback) output to the address of a word that needs to be read for a future calculation. This word is a constant offset away from the original address (depending upon pipelining and other factors) and is output as part of the write operation. If correctly implemented, the exploitation of read-before-write behavior does not affect the recurrence equation of the RNG since both near and far recurrences are constant *relative* offsets themselves.

- *DSP logic blocks:* Many contemporary FPGAs include a certain number of specialized blocks targeted toward DSP multiply/add operations. These blocks may be used to calculate the multi-bit word-wide operations in the RNG recurrence (although the specific implementations in this paper do not do so). The DSP blocks may be more speed-efficient than equivalent logic, and if thse DSP blocks would be unused otherwise, certainly more area-efficient as well.

Two Parallelization Methodologies

We formulate two different methodologies for parallelizing long-period RNGs in hardware: (1) *Interleaved Parallelization* (IP) and (2) *Chunked Parallelization* (CP). Constraints on the degrees of parallelization possible for both of these differ depending on the characteristics of the RNG and result in varying trade-offs between throughput (clock frequency $\times$ degree of parallelization) and area (resource usage). We discuss the methodologies, their constraints and analyze the trade-offs.

4.2 Interleaved Parallelization (IP)

Since both near and far recurrences occur at constant offsets, we can generate more than one random number in parallel if the N-word state vector is interleaved across multiple memory banks. Consider interleaving an RNG with an N-word state vector across a set of banks $\{b_i : 0 \le i < \beta\}$. There are two possibilities for the number of banks; β is either a factor of N, i.e. $N \bmod \beta = 0$ and all banks contain N/β words; or, it is *not* a factor of N and at least one bank contains a fewer number of words than the rest. In the latter case, any recurrence addresses $[(j + M_i) \bmod N]$ that initially point to such a bank for $j + M_i < N$ will point to a different bank for $j + M_i \ge N$.

Interleaved parallelization is inefficient when the number of banks β is not a factor of N because of the additional conditional routing logic (e.g. multiplexers) required. The number of

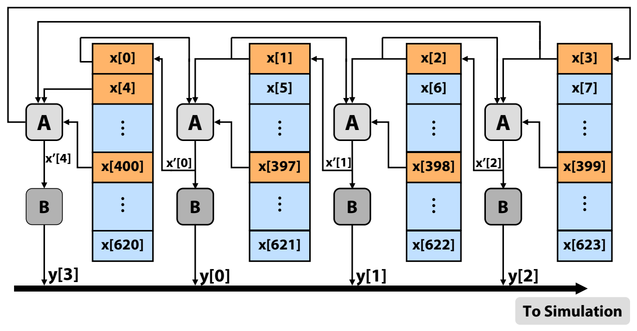


Figure 3: 4-Interleaved Parallelization for `MT19937`

(efficient) interleaved parallelizations possible is therefore the set $\{\beta : \beta$ is a factor of $N\}$. Then, each bank b_i $(0 \le i < \beta)$ contains the state-vector word indices satisfying $j \bmod \beta = i$, while the corresponding recurrences are found in banks corresponding to $[(j + M_i) \bmod N] \bmod \beta$. Asymptotically, as $\beta \to N$, the parallelization approaches that of the N individual registers mentioned in section 3.2 and implemented in [6]; the increase in throughput is not proportional to the increase in area and routing complexity.

For a long-period RNG with w-bit words and q *far* recurrences, assume that the word to be updated is itself included in the recurrence equation and all near recurrences are buffered. If this RNG is interleave-parallelized across β banks and each bank has d v-bit wide read/write operations available per cycle, the number of random words τ output per cycle by this configuration is

$$\tau = \beta \frac{vd}{wq(1+q)} \tag{11}$$

Fig. 3 shows a 4-interleaved parallelization for `MT19937` where $N = 624$ and each 32-bit word $\mathbf{x}[j]$ depends on itself, the near recurrence $\mathbf{x}[j+1]$ and one far recurrence $\mathbf{x}[j+M]$ (all modulo N; $M = 397$). Note that the use of read-after-write or specific pipelining is not shown for clarity. $q = 1$, and if each bank has exclusive use of one dual-port block RAM, from (11), $\tau = 4$ random 32-bit outputs are generated per clock. These are illustrated as $\mathbf{y}[0]$, $\mathbf{y}[1]$, $\mathbf{y}[2]$ and $\mathbf{y}[3]$.

Thus, interleaved parallelization of a long-period RNG can be extremely flexible and scalable depending on how N factors, and provides a constant number of outputs per clock. The primary trade-off is the amount of memory 'slack' (under-utilization of full block RAM capacity), which increases jointly with the degree of parallelization and the number of far recurrences in the RNG.

4.3 Chunked Parallelization (CP)

Chunked parallelization (CP) is based on buffering/re-using not only the near recurrences like IP, but also one far recurrence to further reduce memory accesses. CP splits sequential sections (chunks) of the state vector across a number of banks to make this possible. In effect, the updated output of one bank is not only fed back to that bank, but also used to compute the recurrence equation for a word in another bank (for which this updated result is a far recurrence). The reduced number of read accesses usually result in CP using fewer block RAMs than the same degree of IP. Fig. 4 illustrates degree-3 chunked parallelization for `MT19937`, where each word $\mathbf{x}[j]$ in the state vector depends on one near recurrence $\mathbf{x}[j+1]$ (buffered; not shown) and only one far recurrence, $\mathbf{x}[(j+M) \bmod N]$. The equivalent pseudo-code is shown in Fig. 5.

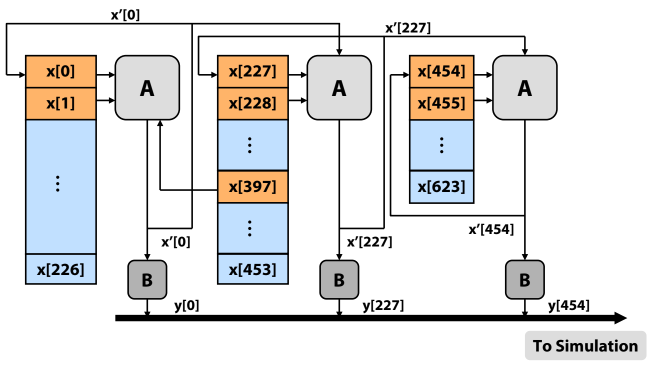


Figure 4: 3–Chunked Parallelization for `MT19937` (case $\mathbf{x}[0] \leftarrow \mathbf{x}[397] \ldots \mathbf{x}[227] \leftarrow \mathbf{x}[0] \ldots \mathbf{x}[454] \leftarrow \mathbf{x}[227]$)

```
A 3-CHUNKED PAR. CONFIG. FOR MT19937()
1  for $j \leftarrow 0$ to $(N - M - 1)$
2  do $\mathbf{x}[j] \leftarrow \mathbf{x}[(j + M) \bmod N]$
3      $\mathbf{x}[j + (N - M)] \leftarrow \mathbf{x}[j]$
4      $\mathbf{x}[j + 2(N - M)] \leftarrow \mathbf{x}[j + (N - M)]$
```

Figure 5: Pseudocode for the 3-CP `MT19937` of Fig. 4

Calculating the number of chunks and chunk lengths (or equivalently, start/end points) is a heuristic process that depends on the position of the far recurrence(s) $\{M_i\}$. Pick one M_i; calculate the set of points $\mathcal{P} = \{(l \cdot M_i) \bmod N\}$ for $l = 0, 1, \ldots, l_{\max} < N$ and sort $\mathcal{P}$ in increasing (or decreasing) order. $\mathcal{P}$ now contains sequential starting points for an $l_{\max}$-chunked parallelization. As $l_{\max} \to (N-1)$, the average length of the chunks in $\mathcal{P}$ approaches $\gcd(N, M_i)$.

For the long-period RNGs we are interested in, M_i are generally primes and very rarely factors of the respective N (except for the special case `MT19937−64`, discussed in section 5.2). This leads to a situation where at least one 'orphan' chunk has a different length than the other (equal) chunks. This phenomenon, with its associated conditional routing overhead and fluctuating output rate, is generally unavoidable for chunked parallelization.

While FIFOs can be used to solve the variable output rate issue, the preferred alternative is to just discard the outputs from the orphan chunk(s). Usually, a CP configuration can be found that leads to relatively large, equal-sized chunks with only one relatively small orphan chunk.

We have written an algorithm that will enumerate $\mathcal{P}$ for various M_i and $l_{\max}$ to deliver one or more 'optimized' CP configurations (such as the one just mentioned). Future work includes deriving analytic expressions (or an algorithm with faster convergence) for optimized CP configurations.

4.4 Interleaved or Chunked?

In general, for a given degree p, a p-IP parallelized configuration has lower routing complexity and higher (or constant-rate) throughput compared to a p-CP configuration; the p-CP on the other hand uses relatively fewer resources than the p-IP. The exceptions to these observations include cases such as:

- If the RNG has a far recurrence that is a factor of N. This occurs for the 64-bit `MT19937−64` ($N = 312$, $M = 156$) and a hybrid 2-CP implementation (discussed in section 5.2) has comparable throughput to a 2-IP implementation while

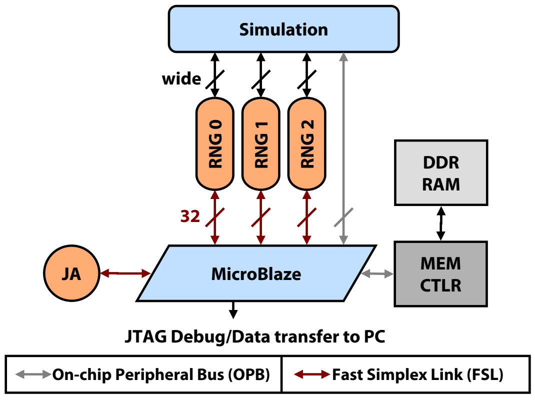


Figure 6: Formalizing the Framework

using fewer resources.

- If conserving block RAM usage is critical (and the variable output rate is not a problem): `MT19937` as 3-CP uses 2 block RAMs and is comparable in throughput to the 3-IP (which uses 3 block RAMs).
- If, for the N-word state-vector of a given long-period RNG, N does not factor into small primes. Consider the `WELL` RNGs [21], which are generalizations of the TGFSR concept. Suppose an application requires a parallelized version of the `WELL44497` RNG (period $2^{44497}-1$; not implemented here), for which the 1391-word state vector N factorizes as 13×107. Therefore, only a 13-IP or 107-IP implementation is possible. If even the 13-IP would be over-engineering for the application, chunked-parallelization must be considered.

  A 6-CP implementation would result in throughput comparable to a (hypothetical) 6-IP, while using less than half the block RAMs and nearly half the logic resources of a 13-IP version. In this case, the 6-CP would consist of six chunks of length 229 each plus an 'orphan' of length 17, whose output is discarded. Note that such discarding is not done in the 3-CP illustration of Fig. 4 because its orphan chunk is relatively large; a FIFO must be used if compensation for the variable output rate is necessary.

Thus, if the RNG state-vector length N factors into small prime numbers, an IP implementation is usually the best choice. If it does not, or if an exceptional situation such as those just discussed arises, CP implementations should be considered.

4.5 Formalizing the Framework

The output(s) of IP/CP long-period RNGs are directly connected in parallel via the logic fabric to the destination application on the FPGA to achieve the highest possible throughput. However, an integrated multiple-stream RNG and parallelized application setup also requires the following essential functions without including dedicated circuitry for each.

1. Seed RNGs when necessary.
2. Efficiently perform the state-vector reads, XORs and write-backs needed for fast jump-ahead.
3. Provide a large secondary memory to store data, e.g. simulation results.
4. Allow control of the RNGs and/or application(s) via a PC; allow data transfer to a PC.

Table 1: Properties of linear recurrences for three Mersenne Twisters: 32-bit (`MT-32`), 64-bit (`MT-64`) and 128-bit (`SFMT`)

RNG	MT-32	MT-64	SFMT
k	19937	19937	19937
w	32	64	128
State N	624	312	156
Near rec.	1	1	2
Far rec.	1	1	1
Elementary Operations for Recurrence			
XOR/Shift/Mask	6/3/12	6/3/12	4/10/4
Elementary Operations for Tempering			
XOR/Shift/Mask	4/4/2	4/4/3	N/A

For these, we complete the framework by adding a soft microprocessor (*MicroBlaze* for Xilinx) to the FPGA running the RNGs and the application(s) as shown in Fig. 6. Each parallelized RNG as well as the application/simulation is connected via 32-bit, point-to-point *Fast Simplex Links* (FSL) to a MicroBlaze processor. The FSL endpoint on the RNG side is a simple finite state machine (FSM) that can pause/continue the RNG and perform state vector reads/writes. Unless a specific seed or seed vector is programmed, the processor initializes the state of each RNG (through the FSM) with the multiplicative congruential generator from [5] that is also used in the `MT19937`-reference code [18]. We note that any user-provided seed or seed-vector *must* be sufficiently random (i.e., roughly equal numbers of `1`s and `0`s). If this is not the case, the state vector starts out in an 'excess-zero' condition and the output of the RNG will be biased (i.e., a higher number of `1`s over `0`s, or vice versa) until it 'recovers' (becomes statistically random) after a certain number of cycles.

The *JA* (Jump-Ahead) unit connected to the MicroBlaze consists of k-bit storage in block RAM and wide XORs for 'accumulating' the intermediate state vectors when jumping an RNG ahead. A 256 MB DDR DRAM module is attached to the MicroBlaze via a DDR controller core on the *On-chip Peripheral Bus* (OPB) and provides storage for simulation results or debug data. Finally, FPGA/MicroBlaze programming and control/data transfer to a PC are via the built-in JTAG interface.

5. IMPLEMENTATIONS

We have implemented three different long-period RNGs in various interleaved and chunked parallelization configurations as well as a multiple stream configuration with 6 copies of one RNG and jump-ahead. Table 1 lists the characteristics of these RNGs including length of the state vector k (the period is $2^k - 1$), native word width w, the state size N (in $w - bit$ words), the number of near/far recurrences and the number of elementary word-width binary operations necessary to compute the recurrence equation as well as that for the 'output transformation/tempering' (if any).

The implementations were targeted to the Xilinx *Virtex-II Pro* XC2VP30-7 (hosted on the *XUPV2P* evaluation board). The designs were coded in VHDL and synthesized with Synplify Pro 9.0.1. The initial designs only had as many pipeline registers/delays as necessary to ensure state consistency for the given parallelized configuration. Compiled designs were then simulated in Modelsim 6.2c and timing analysis performed for further optimization.

Table 2: Various Interleaved and Chunked Parallelizations for the 32-bit Mersenne Twister MT19937

Par. Type	None	2-IP	3-IP	4-IP	8-IP	3-CP	11-CP
Slices	78	159	222	290	566	207	1038
18k BRAMs	2	2	3	4	8	2	7
Freq (MHz)	348.4	349.4	265.1	277.7	283.5	258.3	235.7
Thruput (Gbps)	11.15	22.36	25.45	35.54	72.58	24.03	82.96

Table 3: Selected parallelizations for other long-period RNGs

RNG	MT19937-64			SFMT19937	
Par. Type	None	2-IP	2-CP	None	5-CP
Slices	218	269	246	155	735
18k BRAMs	2	4	2	4	16
Freq (MHz)	339.8	342.9	333.3	260.5	168.0
Thruput (Gbps)	21.75	43.90	42.67	33.34	107.51

5.1 32-bit Mersenne Twister (MT19937)

We have used MT19937 [18] as the long-period RNG representative of its class while discussing the mathematical background and the structure of the parallelization framework. It is used extensively for software simulation and is the only long-period RNG for which published hardware implementations exist (for benchmarking). Therefore, we have implemented a non-parallelized version, four interleaved (of degrees 2, 3, 4 and 8) and two chunked (degrees 3 and 11) parallelized configurations for MT19937 .

Table 2 shows the resource usage and maximal performance statistics for each of these configurations.

5.2 64-bit Mersenne Twister (MT19937-64)

MT19937−64 [20] is a 64-bit version of MT19937 whose recurrence leads to an interesting optimization for a 2-CP implementation. MT19937−64 has an $N = 312$ word state and each word $\mathbf{x}[j]$ depends only upon one near ($\mathbf{x}[j+1]$) and one far ($\mathbf{x}[(j+156) \bmod 312]$) recurrence. $156 = 312/2$, and therefore the far recurrence for $\mathbf{x}[0]$ is $\mathbf{x}[156]$, and the far recurrence for $\mathbf{x}[156]$ is the (updated) $\mathbf{x}[0]$. Chunked parallelization can now be implemented using only one I/O operation per random number generated; also, only two 18-kilobit dual-port block RAMs are necessary since the 64-bit words can be split over two 32-bit ports. The results in table 3 show that this 2-chunked implementation uses half the block RAMs, while maintaining throughput comparable to a standard 2-interleaved parallelization.

5.3 128-bit SIMD-oriented Fast Mersenne Twister (SFMT19937)

Introduced very recently in [24], SFMT is a Mersenne Twister variant optimized for CPUs with instruction sets that support 128-bit Single-Instruction Multiple-Data (SIMD) vector operations, such as SSE/SSE2 (x86) and Altivec (PowerPC). While [24] discusses an SFMT with the standard period $2^{19937}-1$ (SFMT19937), code is available for longer periods up to $(2^{216091}-1)$. SFMT19937 is reported to be between 2–4× faster than MT19937 in software. Since single-cycle 128-bit accesses need 4×32-bit block RAM ports (with a lot of 'slack' memory), we have only implemented a 5-CP version as proof of concept.

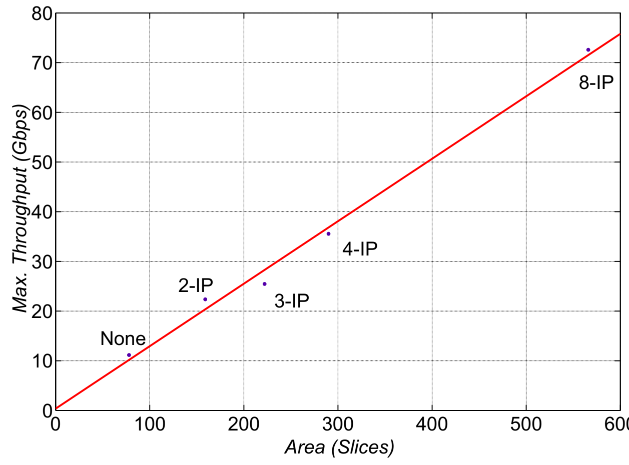


Figure 7: Throughput vs. Area for various MT19937 Interleave-Parallelized (IP) Implementations

5.4 Jump-ahead with six MT19937 RNGs

The fast jump-ahead technique for generating multiple independent sub-streams [3] and the overall framework was tested as proof-of-concept by implementing six identical 2-IP MT19937 PRNGs on the Virtex-II Pro FPGA and setting their initial states apart by 2^{1000}. The auxiliary polynomial $g(z)$ was calculated on a PC using the *Number Theory Library*[2] (NTL) and, along with the jump-ahead procedure, programmed into the MicroBlaze.

Each jump (5 total) took approximately 36.113 ms, with the 2-IP MT19937s clocked at 250 MHz and the MicroBlaze processor/Fast Simplex Link bus at 100 MHz. This includes the overhead of transmitting each state corresponding to a non-zero coefficient in $g(z)$ to the MicroBlaze (via the FSL link) and accumulating it in the jump-ahead (JA) unit, as illustrated in Fig. 6. The average times for a large number of random jump-aheads with MT19937, as described in [3], is 15.9 ms for a 32-bit Pentium 4 and 9.0 ms for a 64-bit Athlon processor; these are not directly comparable to our proof-of-concept FPGA implementation.

We expect jump-ahead performance to improve as this initial implementation is optimized, e.g. by increasing the bandwidth of the RNG⟷MicroBlaze link.

5.5 Analyses and Benchmarks

To our knowledge, the fastest FPGA implementations of any long-period RNG reported in the literature are the MT19937 in [6], on an Altera Stratix FPGA. We believe this is a fair comparison since both the Altera Stratix and the Virtex-II Pro are based on a 1.5V, 0.13µm process. As shown in Table 2, the average throughput of the 2-IP, 4-IP and 8-IP parallelized MT19937 implemented

[2] *http://www.shoup.net/ntl/*

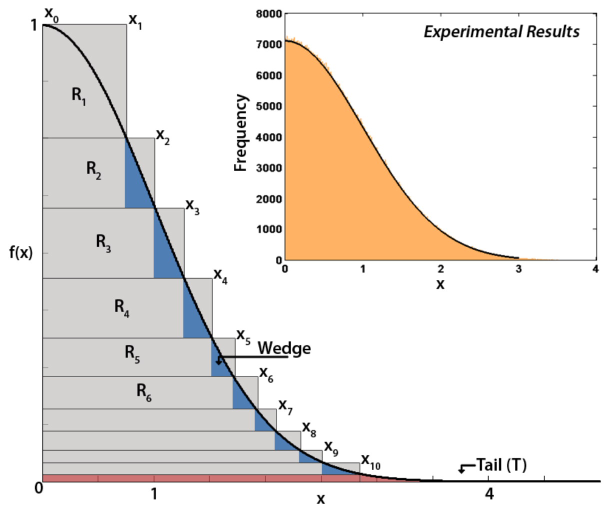


Figure 8: Ziggurat partitioning of the normal distribution with $N = 10$ rectangles and the tail T. Inset: Output from FPGA implementation

on the Virtex-II Pro is $\approx 4.8\times$ that of their equivalents in [6], while using between 21% to 41% less logic (assuming a Xilinx slice ≈ 2 Altera logic elements). Since [6] only provides the actual number of memory bits used and not the number or utilization of block memories, we cannot make a direct comparison; however, since the structure they describe requires pseudo-quad-ported memories, probably synthesized from multiple dual-port blocks with data duplication, we estimate they would use at least double the number of block memories for similar degrees of parallelization.

As another comparison, the 2-IP implementation of MT19937 is 7.1× faster (22.36 Gbps) than an experimental run of the equivalent C-code on a Pentium 4 processor at 3 GHz (3.14 Gbps at 100% utilization).

Fig. 7 plots throughput vs. area for the 2-, 3-, 4- and 8-IP implementations shown in Table 2, along with a linear least-squares fit. The throughput/area efficiency for IP remains constant with increasing degrees of parallelization. This does not seem to be the case for chunked-parallelization, based on the two-data points available (Table 2: 3-CP, 11-CP); CP efficiency decreases with scaling.

If a specific long-period RNG is not mandated by an FPGA application, our recommendation is to use the 2-chunk-parallelized MT19937−64, which has the highest overall efficiency [i.e., max. throughput/(slices×BRAMs)] of all RNGs implemented.

6. APPLICATIONS

Two different applications that require multiple random number streams at varying rates have been implemented on the Virtex-II Pro and coupled to the RNGs to test our integrated framework. The applications have been pipelined and parallelized whenever obvious. In the following sub-sections we briefly discuss the theory behind each application and present notable observations about its implementation.

6.1 The Ziggurat Method for generating normal Random Variables

Probably the most common non-uniform distribution that applications require random numbers from is the normal (or Gaussian) distribution. An efficient approach for generating normal random variables from uniform random numbers is the *Ziggurat* method [15]. This is an acceptance/rejection method; given a set $\mathcal{C}$ for the target distribution (normal), it picks *random* points from a discrete superset $\mathcal{B}$ of $\mathcal{C}$ and checks to see if they are also in $\mathcal{C}$ before keeping or discarding them.

The set $\mathcal{C}$ of points (x, y) that make up the area of a decreasing function $f(x)$ such as the normal distribution can be defined as $\mathcal{C} = \{(x, y) : y \leq f(x)\}$. The Ziggurat method partitions the normal distribution into N rectangles of equal area V to define a superset $\mathcal{B} \supset \mathcal{C}$ as

$$\mathcal{B} = \left(\bigcup_{i=1}^{N} R_i\right) \cup T$$

where R_i $(0 < i \leq N)$ are the rectangles extending from $x = 0$ to $x = x_i$ and $y = f(x_i)$ to $y = f(x_{i-1})$ and T is the base rectangle plus the tail area from $x = x_0$ to $x = \infty$. Fig. 8 shows an example with $N = 10$ rectangles overlaid on a plot of the normal distribution for $x \geq 0, \mu = 0$ and $\sigma^2 = 1$.

The method can be summarized as:

1. Draw a random uniform number u
2. Choose a random index i with probability $1/N$ and assign u to rectangle R_i by scaling it to $x = ux_i$.
3. If $0 < i \leq N$, check if $x < x_{i-1}$ (i.e., does $x \in \mathcal{C}$—is it under the curve of $f(x)$?). If yes, return x; **else**
4. Generate a random point y on the line segment $y_{i-1} \rightarrow y_i$ and check if $f(x) \geq y$ (i.e., is x in the *wedge* area of rectangle R_i (see fig. 8)). If yes, return x.
5. If $i = 0$, generate an x from the tail as described in [13] (this involves at least one iteration of calculating the natural logarithms of two uniform random numbers).

The efficiency of the algorithm is proportional to the number of rectangles N; for $N = 128$, it has a success rate of 98.78%, and terminates at step 3 98.05% of the time (so that no log or exponential calculations are required). In practice, the algorithm is sped up by using pre-computed lookup tables for the static values of x_i and $f(x_i)$ used in comparisons.

We implemented the algorithm for $N = 128$. The chi-square (χ^2) goodness-of-fit test was applied to a 3-million point output data-set (inset of Fig. 8) and gave a χ^2 statistic of 298.62 and $p = 0.982$ affirming the quality of the normal random variables generated.

6.2 Monte Carlo: Light Propagation in a Scattering Medium

Monte Carlo methods use stochastic models to simulate the behavior of a process. We simulate light propagation/heat transfer in a scattering medium (such as tissue) by a point source [23]. As photons are absorbed, their location and the heat transferred are recorded in bins in an absorption matrix corresponding to spatial locations in the medium. The model is simplified and does not include internal reflections.

Table 4: Applications: Resources and Performance

App.	Ziggurat	Monte Carlo
Slices	1,227	2,466
18k BRAMs	2	4
18 × 18 Mult	21	64
Freq (MHz)	60	51

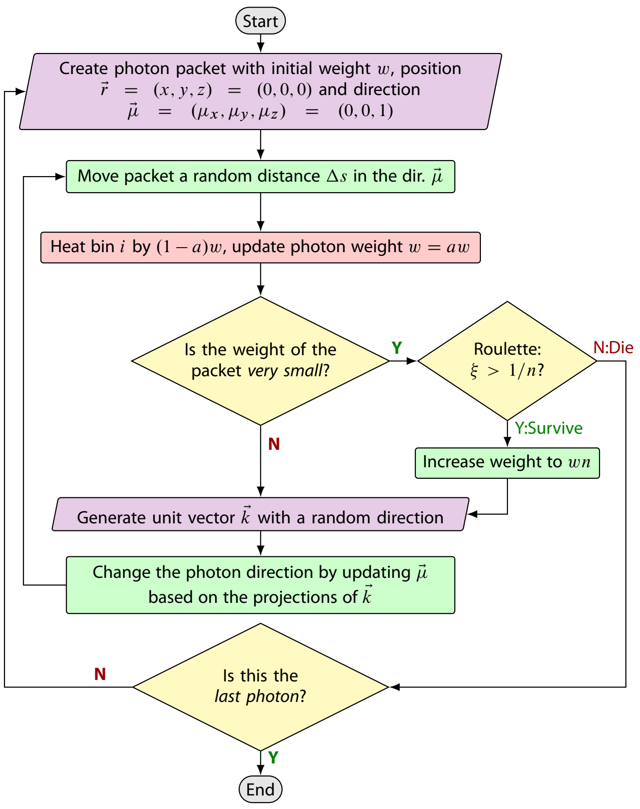
Fig. 6.2 presents a flowchart for the simulation. Photon packets with initial weight w are created, injected into the medium and propagated further by random distances Δs. Δs is calculated as $-\ln \xi/\mu_t$, where ξ is a uniform random variable and μ_t depends on the absorption and scattering coefficients of the medium. At the end of each propagation step, a fraction of the photon packet 'splits' and is absorbed into a bin (calculated from the position vector).

If the new weight of the remainder is 'significant', it is scattered in the direction of a random unit vector $\vec{k}$ and propagated for another step. However, if the weight is very small, the packet goes into a *roulette* and its survival is determined by comparing a random variable to a given chance of survival. If the photon packet survives, its weight is increased and it is scattered and propagated further; if not, the process starts all over with a new photon.

Calculating the unit vectors for direction changes and the norm of the position vector (to determine the absorption bin) requires a square-root, a divider and a number of multiplies leading to the large logic usage in Table 4. A Monte Carlo simulation for 10^6 photons required approximately 141×10^6 random numbers and its results are in agreement with the analytical diffusion equation for a point source.

7. STATISTICAL TESTING AND CONCLUSION

The word order in the output sequence of an interleave-parallelized (IP) or chunk-parallelized (CP) long-period RNG is permuted compared to the sequential/software implementation.The CP-versions may also discard portions of the sequence based on their design. While neither parallelized implementation modifies



Start
Create photon packet with initial weight w, position $\vec{r} = (x,y,z) = (0,0,0)$ and direction $\vec{\mu} = (\mu_x,\mu_y,\mu_z) = (0,0,1)$
Move packet a random distance Δs in the dir. $\vec{\mu}$
Heat bin i by $(1-a)w$, update photon weight $w = aw$
Is the weight of the packet *very small*?
Y
Roulette: $\xi > 1/n$?
N:Die
Y:Survive
Increase weight to wn
N
Generate unit vector $\vec{k}$ with a random direction
Change the photon direction by updating $\vec{\mu}$ based on the projections of $\vec{k}$
N
Is this the *last photon*?
Y
End

Figure 9: Monte Carlo simulation for light propagation in a scattering medium

the linear recurrence equation defining the RNG, we subjected the output of all the implementations in Tables 2 and 3 to two standard statistical test batteries for empirically estimating random number quality: *Diehard* [14] and *Crush* (from TestU01 [11]).

The IP– and CP–RNGs passed all the *Diehard* tests with p-values between 0.0194 and 0.9882. All the tests in the *Crush* battery were passed except for the linear-complexity test for degrees $n > k$, where k is the length of the state-vector for the RNG in bits. Every RNG defined as a linear recurrence will fail this test since any bits following a $k-$bit sub-sequence can be expressed as linear combinations of those previous k bits. As expected from the theory, neither IP- nor CP-parallelizations have had any effect on the inherently high quality of the random numbers generated by the long-period RNGs.

Conclusion

We have developed a hardware framework for generating multiple independent random number streams from 'long-period' pseudo-random number generators such as the *Mersenne Twister*. Within the context of this framework, we presented two general architectures for parallelizing long-period RNGs in hardware. The architectures exploit the properties of the linear recurrence defining the RNGs to minimize I/O accesses from block RAM and thus maximize throughput.

We have also implemented various configurations of three different variants of the Mersenne Twister on the Xilinx Virtex-II Pro FPGA. To our knowledge, based on a review of the literature, one is the fastest existing hardware implementation of the 32-bit Mersenne Twister while the other two have been implemented in hardware for the first time. A proof-of-concept implementation for the recently proposed 'fast-jump ahead' technique of advancing an RNG is also discussed, thus allowing multiple independent, non-overlapping sub-streams to be generated.

Finally, we complete our framework by testing the RNGs with two different applications: the Ziggurat method for generating normal random variables and a Monte Carlo simulation for photon-transport. Future work includes extending the framework the to the WELL family of RNGS, optimizing it for newer FPGA architectures (such as the Virtex-5) and improving the speed of the jump-ahead technique.

The availability of such a hardware framework and fast parallelized implementations of well-proven long-period RNGs with multiple streams can accelerate existing hardware-based scientific applications and simulations and also encourage parallelized FPGA implementations of complex simulations that are currently software-only.

Acknowledgments

The authors would like to thank the anonymous reviewers, Fred Fontaine, Jared Harwayne-Gidansky and David Nummey for their insightful comments and suggestions. Christopher Mitchell's help with some of the figures is also appreciated.

8. REFERENCES

[1] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte carlo simulations: Hidden errors from ''good'' random number generators. *Phys. Rev. Lett.*, 69(23):3382–3384, Dec 1992.

[2] J. E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, 2003.

[3] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for $\mathbf{F}_2$-linear random

number generators. *INFORMS Journal on Computing*, 2007. to appear.

[4] B. Jun and P. Kocher. *White Paper: The Intel Random Number Generator*. Cryptography Research, Inc., 1999.

[5] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison Wesley, Reading, Mass., third edition, 1997.

[6] S. Konuma and S. Ichikawa. Design and evaluation of hardware pseudo-random number generator MT19937. *IEICE Trans. on Info. and Systems*, 88(12):2876–2879, 2005.

[7] T. Kurokawa and H. Kajisaki. FPGA based implementation of Mersenne Twister. *Sci. Eng. Rep. Nat. Def. Acad. (Japan)*, 40(2):15–21, Mar. 2003.

[8] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw Hill, New York, NY, third edition, 2000.

[9] P. L'Ecuyer and F. Panneton. Construction of equidistributed generators based on linear recurrences modulo 2. In *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 318–330, Berlin, 2002. Springer-Verlag.

[10] P. L'Ecuyer and R. Simard. On the performance of birthday spacings tests with certain families of random number generators. *Math. Comput. Simul.*, 55(1-3):131–137, 2001.

[11] P. L'Ecuyer and R. Simard. Testu01: A software library in ANSI C for the empirical testing of random number generators, 2003.

[12] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, 20(3):456–468, 1973.

[13] G. Marsaglia. Generating a variable from the tail of the normal distribution. *Technometrics*, 6(1):101–102, 1964.

[14] G. Marsaglia. The Diehard battery of tests of randomness, 1995.

[15] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *J. Statistical Software*, 5(8):1–7, 2000.

[16] J. L. Massey. Shift-register synthesis and bch decoding. *IEEE Trans. Info. Theory*, 15:122–127, 1969.

[17] M. Matsumoto and Y. Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266, 1994.

[18] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, Jan. 1998.

[19] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1992.

[20] T. Nishimura. Tables of 64-bit Mersenne Twisters. *ACM Trans. Modeling and Computer Simulation*, 10(4):348–357, Oct. 2000.

[21] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw.*, 32(1):1–16, 2006.

[22] A. S. Pasciak and J. R. Ford. A new high speed solution for the evaluation of monte carlo radiation transport computations. *IEEE Trans. Nuclear Science*, 53(2):491–499, Apr. 2006.

[23] S. A. Prahl, M. Keijzer, S. L. Jacques, and A. J. Welch. A Monte Carlo model of light propagation in tissue. In *SPIE Proc. of Dosimetry of Laser Radiation in Medicine and Biology*, volume IS5, pages 102–111, 1989.

[24] M. Saito. An application of finite field: Design and implementation of 128-bit instruction-based fast pseudorandom number generator. Master's thesis, Hiroshima University, Feb. 2007.

[25] L. Salwinski and D. Eisenberg. In silico simulation of biological network dynamics. *Nature BioTech.*, 22:1017–1019, 2004.

[26] A. Srinivasan, M. Mascagni, and D. Ceperley. Testing parallel random number generators. *Parallel Computing*, 29(1):69–94, 2003.

[27] V. Sriram and D. Kearney. An area time efficient field programmable Mersenne Twister uniform random number generator. In *Proc. Int. Conf. on Eng. of Reconfigurable Systems & Algorithms*, pages 244–246, 2006.

[28] R. C. Tausworthe. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19:201–209, 1965.

[29] D. B. Thomas and W. Luk. High quality uniform random number generation using LUT optimised state-transition matrices. *J. VLSI Sig. Proc.*, 47(1):77–82, Apr. 2007.

APPENDIX

A. THE MATHEMATICS OF JUMP-AHEAD

We briefly elaborate on the math behind the fast jump-ahead discussed in section 3.3, as proposed in [3]. From the Cayley-Hamilton theorem, the recurrence matrix $\mathbf{A}$ in (6) satisfies its own characteristic equation, i.e.

$$p(\mathbf{A}) = \mathbf{A}^k + a_1\mathbf{A}^{k-1} + \cdots + a_{k-1}\mathbf{A} + a_k\mathbf{I} = 0 \tag{12}$$

Let a polynomial

$$g(z) = z^\nu \bmod p(z) = \alpha_1 z^{k-1} + \cdots + \alpha_{k-1} z + \alpha_k, \tag{13}$$

which can be pre-computed in $O(k^2 \log \nu)$ time [5]. Also, $g(z) = z^\nu + p(z)q(z)$ for some $q(z)$; since $p(\mathbf{A}) = 0$,

$$g(\mathbf{A}) = \mathbf{A}^\nu = \alpha_1\mathbf{A}^{k-1} + \cdots + \alpha_{k-1}\mathbf{A} + \alpha_k\mathbf{I}$$

and the new state (10) can be computed as

$$\begin{aligned} \mathbf{A}^\nu\mathbf{x} &= (\alpha_1\mathbf{A}^{k-1} + \cdots + \alpha_{k-1}\mathbf{A} + \alpha_k\mathbf{I})\mathbf{x} \\ &= \mathbf{A}(\ldots\mathbf{A}(\mathbf{A}(\mathbf{A}\alpha_1\mathbf{x} + \alpha_2\mathbf{x}) + \alpha_3\mathbf{x}) + \ldots \\ &\quad + \alpha_{k-1}\mathbf{x}) + \alpha_k\mathbf{x} \end{aligned} \tag{14}$$

by the standard Horner's method for polynomial evaluation. Effectively, the RNG is run for $(k-1)$ steps and the states (k-bit vectors) corresponding to the non-zero α_i in (14) are XORd together. While $O(k^2)$ bitwise operations are still needed, memory requirements are reduced to k-bits of storage for the accumulator.