FPGA-Optimised High-Quality Uniform Random Number Generators

David Barrie Thomas
Imperial College
London, England
dt10@doc.ic.ac.uk

Wayne Luk
Imperial College
London, England
wl@doc.ic.ac.uk

ABSTRACT

This paper introduces a method of constructing random number generators from four of the basic primitives provided by FPGAs: Flip-Flips, Lookup-Tables, Shift Registers, and RAMs. The construction method is designed to ensure maximum clock rates, while using the minimum of resources, and providing statistical quality at the level of the best software generators. In all platforms tested, the generators are limited in speed only by the clock distribution network or the maximum clock speed of the underlying RAM primitives, using a platform independent VHDL description with no placement or other hints. The area utilisation is also very low, with a Virtex-5 generator requiring just one Block-RAM and 41 slices to produce 48Gb/s at 550MHz: over 14 times faster than the commonly used Mersenne-Twister RNG on an Opteron at 2.2GHz, while providing the same level of quality.

Categories and Subject Descriptors

B.6.0 [**Hardware**]: Logic Design—*General*

General Terms

Algorithms,Design,Performance

Keywords

Random Number Generation

1. INTRODUCTION

Many scientific and industrial problems have no tractable closed-form solution, and can only be solved through Monte-Carlo simulations. However, such simulations require huge amounts of computational power, and the power and size limits of conventional compute-farms have led to significant interest in the use of FPGAs in such applications.

One problem common to any software or hardware Monte-Carlo simulation is the efficient generation of high-quality random numbers: seemingly trivial statistical flaws in the low-level random input can introduce bias into the overall results of the simulation. The software community has developed a number of high-quality, long period Random Number Generators (RNGs), some of which have been adapted for use in FPGAs. However, these generators were designed to meet the needs of word-level instruction processors, and so are less efficient when mapped to the bit-level operations available in FPGAs.

This paper presents a novel way of creating long period RNGs based on a parametric description of FPGA resources such as block RAMs. The benefits of this approach are:

- RNGs can be customised to meet the requirements of both the target FPGA family and the application, minimising wasted resources found when using just one RNG for multiple architectures and applications.
- The RNGs are constructed in a way that guarantees both performance and efficient use of logic, by controlling the longest signal path, and tailoring the RNG transition function to the LUTs of the target family.
- RNGs with very high statistical quality and long periods can be created, with statistical quality on a par with the best software generators.
- A generator with period $2^{11213} - 1$, high performance, and efficient use of logic is realised in a variety of architectures, with a Virtex-5 generator using one block RAM and 113 LEs providing random bits at 48Gb/s, over 14 times faster than a software generator of equivalent quality on an Opteron at 2.2GHz.

We first provide an overview of RNGs, and some methods for implementing RNGs that have been used in previous work. The strengths and weaknesses of previous approaches are then used to define a set of seven performance and quality requirements for any RNG intended for use in Monte-Carlo simulations. Section 4 then presents our architecture-independent methodology for constructing RNGs that guarantees performance and efficient use of logic, followed by a description of the search process that can search within these RNGs for those with maximum period and good statistical quality in Section 5. Practical results for a number of RNGs are presented in Section 6, followed by a comparison of the RNGs proposed here and previous work.

2. MOTIVATION

There are two types of RNG: True (TRNG), and Pseudo (PRNG). TRNGs make use of physical phenomena, for example the jitter between oscillators [2] to produce numbers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'08, February 24–26, 2008, Monterey, California, USA.
Copyright 2008 ACM 978-1-59593-934-0/08/02...$5.00.

that are fundamentally unpredictable, and are of great interest in cryptographic applications. However, such generators cannot produce high bit-rate streams of random bits, and it is impossible to repeat a simulation run with the same stimuli without storing the entire sequence of bits, so such generators are not appropriate for Monte-Carlo simulation. This paper focusses exclusively on PRNGs, and we will use the terms PRNG and RNG interchangeably.

A Pseudo-Random Number Generator (PRNG) can be described using a tuple $(\mathbf{S}, f, \mathbf{R}, r)$, where: $\mathbf{S}$ is the state space of the generator; $f : \mathbf{S} \mapsto \mathbf{S}$ is a function mapping one generator state to another state; $\mathbf{R}$ is the set of random outputs that can be generated; and $r : \mathbf{S} \mapsto \mathbf{R}$ is a function that extract a random output from a given generator state. A random stream $\mathbf{x}_1, \mathbf{x}_2, .. \in \mathbf{R}^\infty$ is generated by choosing an initial state $\mathbf{s}_0 \in \mathbf{S}$, then advancing the generator state using the recurrence $\mathbf{s}_{i+1} = f(\mathbf{s}_i)$, and mapping each state to a random output in the stream with $\mathbf{x}_i = r(\mathbf{s}_i)$.

An important property of a pseudo-random generator is that f and r are deterministic functions: a given seed state $\mathbf{s}_0$ must always give rise to the same output sequence. This allows a simulation to be restarted from a known state, for example to verify an interesting result, or to continue a simulation run that had to be aborted.

A second property is that for practical purposes $\mathbf{S}$ must have a fixed cardinality, as a computer cannot efficiently store and manipulate an unbounded state-space. A direct consequence is that a PRNG must eventually repeat, as eventually the transition function will return to the initial state $\mathbf{s}_0$. The period of a PRNG from a given seed can be defined as $p(\mathbf{s}_0) = \min_{t>1} : \left[f^t(\mathbf{s}_0) = \mathbf{s}_0\right]$, with the upper bound of $p(\mathbf{s}_0) \leq |\mathbf{S}|$. The period is seed-dependent because f may exhibit multiple distinct cycles in the state-space.

Consider $\mathbf{S} = [0..8)$ and $f(s) = (s + 2 \mod 8)$. The seed 0 leads to the state sequence $0, 2, 4, 6, 0, 2...$, while the seed 1 provides $1, 3, 5, 7, 1, 3...$, so there are two cycles of period 4. However, if $f(s) = (s + 3 \mod 8)$, then there is only one cycle $0, 3, 6, 1, 4, 7, 2, 5, 0$ of length 8. A key goal when designing a PRNG is to guarantee that a given seed will result in a long period, as close to the upper bound as possible.

Although there are many methods for generating random numbers, the most commonly used tend to fall into two broad groups of linear generators, with the difference being whether they operate on integers or bits. Obviously in an instruction processor both are implemented using word-based instructions, but the distinction is between integer level instructions such as addition and multiplication, and instructions such as bit-wise xor and shifting.

Bit-level generators are of much more interest for FPGAs, as bit-level processing is a native processing feature, while in CPUs it must be implemented using multiple word-width bit-wise operations. The majority of bit-level generators use binary linear recurrences, where each bit in the next state is formed from the exclusive-or of one or more bits in the previous state.

2.1 Binary Linear Recurrences

A linear generator contains an n bit state, and produces w bit outputs.

$$f(\mathbf{s}_i) = \mathbf{s}_{i+1} = \mathbf{A}\mathbf{s}_i, \qquad \mathbf{s}_i = (s_{i,1}, ..., s_{i,k})^{\mathrm{T}} \in \mathbb{F}_2^n \tag{1}$$

$$r(\mathbf{s}_i) = \mathbf{x}_{i+1} = \mathbf{B}\mathbf{s}_i, \qquad \mathbf{x}_i = (x_{i,1}, ..., x_{i,w})^{\mathrm{T}} \in \mathbb{F}_2^w \tag{2}$$

The matrix $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{B}$ is a $w \times n$ matrix, with all elements in $\mathbb{F}_2$. A generator has the maximum period of $2^n - 1$ iff the characteristic polynomial $P_z = \det(\mathbf{A} - z\mathbf{I})$ has degree n and is primitive over $\mathbb{F}_2$. In the special case that $2^n - 1$ is a (Mersenne) prime it suffices to show that P_z is irreducible and of degree n.

The characteristic polynomial directly determines the values of bits within the state sequence. Given a recurrence of length n, its maximum period characteristic polynomial $P_z = z^n - a_1 z^{n-1} + ... + a_n z^0$, and the values of one bit position j within n successive states, the value of the bit position within the next state can be calculated as:

$$s_{n,j} = a_1 s_{n-1,j} \oplus a_2 s_{n-2,j} \oplus ... \oplus a_{k-1} s_{n-1,j} \oplus a_k s_{n-k,j} \tag{3}$$

The same relation holds for all bits within the state, as each bit follows the same repeating sequence of period $2^n - 1$. Although this introduces a direct relationship between different state bits, one can construct the recurrence so that the offsets between the streams are all extremely long.

In addition, the number of non-zero coefficients in P_z can be viewed as the number of taps in a shift-register of length n. The minimum number of taps is two, with one tap at n and one at some value $1 < l < n$, however, this introduces obvious correlations between the current state and the states observed l and n steps ago. Thus a well-known heuristic for achieving good statistical randomness is to ensure that P_z contains an approximately equal number of zero and non-zero coefficients.

We will now briefly describe some types of recurrence that have been used in existing FPGA generators, before moving onto the new families of binary linear recurrence introduced by this paper, in the following section.

2.2 LFSR

The LFSR (Linear Feedback Shift Register) is the most direct form of binary linear recurrence, as it simply implements the characteristic polynomial. For example, the polynomial $x^6 + x^5 + x^0$ translates to the state transition function:

$$\mathbf{s}_{i+1}{}^{\mathrm{T}} = \left[s_{i,5} \oplus s_{i,6}\ ,\ s_{i,1}\ ,\ s_{i,2}\ ,\ s_{i,3}\ ,\ s_{i,4}\ ,\ s_{i,5}\right]^{\mathrm{T}} \tag{4}$$

Note that only one bit, $s_{1,i+1}$, represents a "new" value. All the other state bits are simply shifted copies of values from the previous state. So although in principle the LFSR generates n bits per step, only one of them is actually useful.

In the rest of this paper we will use the following terminology to refer to the two kinds of bits: "Active Bits" are bits formed from a combination of two or more bits in the previous state, while "FIFO Bits" are a direct copy of just one bit from the previous state. Only active bits can reasonably be considered as independent random bits, so the maximum number of random bits taken from an RNG per step is the number of active bits.

An LFSR only has one active bit, so to generate w random bits per step it is necessary to use w separate LFSRs, and combine one bit from each [1]. Unfortunately this means that wn bits of storage only produce a sequence of length $2^n - 1$, which is much less than the maximum possible period of $2^{nw} - 1$.

[1] The LFSR presented here is the Fibonacci LFSR, but it is also possible to run the generator in reverse, known as the Galois LFSR. Even though the Galois LFSR has more active bits, they still have significant correlations and are completely unsuitable for use as independent random bits.

2.3 Combined Tausworthe

The Combined Tausworthe is an RNG designed to overcome some of the deficiencies of simple linear recurrences, while retaining an efficient implementation in software. Instead of a single recurrence, the matrix $\mathbf{A}$ is comprised of k separate w-bit recurrences, $\mathbf{M}_1..\mathbf{M}_k$, which are stepped independently then exclusive-or'd together to produce the output sample. The component matrices are chosen so that their periods are relatively prime, resulting in an overall period that is the product of all the individual periods.

The Combined Tausworthe offers good performance in software, but the period is typically much less than $2^{wk}-1$. For example, a common parametrisation called Taus113 [4] provides a period of 2^{113}, much less than the potential period of $2^{128}-1$. However, this generator still offers much better quality than an LFSR, as the characteristic polynomial is relatively dense. The Combined Tausworthe has also seen some use in hardware [6, 15], as it is simple to express, and achieves good clock rates.

2.4 LUT Recurrence

A very efficient method for generating random numbers in FPGAs with periods less than $\sim 2^{1000}$ is provided by the LUT optimised recurrence [13]. This is specifically designed for LUT-based FPGAs (or any bitwise fabric), and has no efficient software implementation. The basic idea is that, to provide maximum area efficiency, each state bit should also be an active bit, as every FF comes with a LUT for free. This can be achieved using a recurrence matrix with t ones in each row, using a random search method to construct candidate matrices and testing the characteristic polynomial of each matrix for primitivity.

We claim this method is at the upper bound on efficiency (in terms of maximising active bits per resource), as it is impossible to construct a generator that has more active bits with lower resource requirements: an n bit LUT optimised generator uses exactly n LUTs and n FFs, and produces n random bits per cycle. The performance is also extremely good, with the clock rate usually limited by the clock distribution network for $n \leq 512$.

However, the big drawback of this method is that to create a long period sequence, a large number of LUT-FF pairs must be used. Even if an application only needs 64 bits per cycle, it must use 512 LUT-FFs to get a period of $2^{512}-1$, so 87% of the performance is wasted. Even when it is acceptable to trade resources for a long period, the search method used to find RNGs becomes computationally infeasible for $n > 1500$, imposing a limit on the maximum period.

2.5 The Mersenne Twister

The Mersenne Twister (MT19937) is an RNG developed for 32-bit CPUs, providing a period of $2^{19937}-1$. It has seen extensive use in a wide variety of applications, and may be seen as the quality benchmark by which other RNGs are measured. Internally MT19937 is just a binary linear recurrence, albeit with a very large recurrence matrix. The generator is one of a family described using the 32-bit word-level recurrence:

$$\mathbf{x}_{i+t} = \mathbf{x}_{i+m} \oplus \mathbf{y}\mathbf{R}, \quad \text{where} \quad \mathbf{y} = \mathbf{x}_t[31] \ \& \ \mathbf{x}_{t+1}[30:0] \quad (5)$$

where $\mathbf{R}$ is a 32×32 matrix, and m and t are generator specific parameters with $0 \leq m < t$. In the parameters used in MT19937, $t = 624$, so the total number of bits in the state is 19968, but because only the MSB of $\mathbf{x}_k$ is used when forming $\mathbf{y}$, the period is $2^{19937}-1$.

One deficiency of MT19937 is that, although the period is very long, the characteristic polynomial only has 135 non-zero terms. This leads to a known weakness, whereby the generator takes many thousands of iterations to escape an initial state with a very low Hamming weight.

MT19937 has been implemented in FPGAs [3], but is limited to generating 32 bits per cycle; if an application requires 33 bits, then two parallel generators must be used, with 31 bits going to waste. The choice of parameters is also inefficient for FPGA block RAMs, as the most common RAM width is 36, so 4 bits (11%) of RAM bandwidth is wasted.

A similar software generator that has been adapted for hardware [11] is TT800 [9], which uses 25 32-bit words to provide a period of $2^{800}-1$, with a much denser polynomial than MT19937.

3. REQUIREMENTS

All the previous generators have a number of strengths and weaknesses, some of which are important in FPGAs, and some of which are not. We now present a set of minimum requirements for a high-quality RNG that can be safely and efficiently deployed in any FPGA-based Monte-Carlo simulation:

1. Period - The RNG must provide sequences with periods of 2^{2048} or more. Long periods make it much easier to manage parallel applications, as independent RNGs can simply be initialised with random states without worrying about potential overlap between sequences.

2. Performance - The RNG must provide the maximum clock rates allowed by a device, without requiring device-specific HDL or floor-planning. Calculations within Monte-Carlo simulations can often be heavily pipelined [14], with the speed limited only by the block RAM and DSP switching rates, so we must ensure the RNG can supply random stimuli at the same rates.

3. Empirical Quality - The RNG should convincingly pass all non-cryptographic empirical tests for randomness. These tests do not *guarantee* statistical quality, but they provide a high degree of confidence.

4. Density - All linear RNGs have a characteristic polynomial, representing the lags from which multiple previous samples are taken to form the next sample. A dense polynomial, with approximately half the coefficients non-zero, is a well known criterion for a good quality generator, as it eliminates the possibility of simple auto-correlations at a small number of lags.

5. Customisation - Can be customised to provide a specific number of random bits, to meet the demands of different applications. Each application requires a different set of random stimulus per cycle, often including non-uniform numbers that are themselves constructed from many uniform bits [12, 6]. A key advantage of FPGA technology is the ability to customise data-path widths, so the RNG must directly support variable numbers of random bits.

6. Memory Efficiency - Take into account the memory architecture of FPGAs to fully utilise RAM bandwidth. The state of long period RNGs can only feasibly be implemented in RAMs, so it is important to make sure that the state is mutated as fast as possible by making use of all available RAM bandwidth.

	LFSR	Taus	LUT-Opt	MT19937	TT800
Period				✓	
Performance	✓	✓	✓		
Empirical Quality	✓		✓	✓	✓
Density		✓	✓		✓
Customisation			✓		
Memory Efficiency	✓				
Logic Efficiency			✓		

Table 1: Summary of how previous RNGs meet the seven requirements.

7. Logic Efficiency - The RNG should efficiently use logic resources, and asymptotically approach the upper limit of one LUT-FF pair per active bit as the number of output bits increases.

Table 1 is an attempt to summarise how the previous generators match our set of requirements. The LUT-optimised generator comes closest, but is let down by its inability to provide long periods. MT19937 is the only long-period generator, but falls short in most other regards. It should be made clear that this is not a condemnation of the MT19937: it's a highly efficient generator, *when used in software*. The other generators also have advantages not considered important here; for example, the Combined Tausworthe can be expressed in just a few lines of C code.

Given the absence of any RNGs meeting the requirements of high-performance high-quality FPGA simulations, we will now present a family of generators that *do* meet the requirements.

4. CONSTRUCTION OF LINEAR RECURRENCES OPTIMISED FOR FPGAS

In this section we introduce a family of RNGs guaranteed to satisfy our requirements on performance, customisation and resource efficiency, while providing the necessary (but not sufficient) conditions for the period and quality requirements to be met. Then, in Section 5, we explain how these properties can be verified for a given member of this family.

4.1 FPGA Building Blocks

For the purposes of building linear recurrences, there are three relevant types of FPGA resource:

1. Basic LUT-FF Logic Element (LE).
2. Shift-Registers.
3. RAMs.

In terms of the two types of bits in a linear recurrence, it is clear that active bits can only be placed in LEs. In principle, FIFO bits could also be placed in an LE, but this would contravene our requirement that we use the minimum number of resources per active-bit, as the LUT in the LE would be wasted. The only time this convention may be violated is if a RAM does not contain an output register, where it is necessary to introduce FFs to meet the stronger requirement on performance. While LEs can be either active or FIFO bits, all other resources can *only* be used as FIFO bits, as they contain no active logic.

We identify Shift-Registers as any component that is able to provide a fixed-length multi-cycle delay. These often consist of an underlying RAM component, plus some hardened FIFO logic, allowing the resource to be used either as a RAM or a FIFO.

However, many hardened FIFO circuits are designed to support the more complex case of independently clocked read/write enables, by using an underlying Dual Port RAM. Unfortunately, this wastes RAM bandwidth when implementing a fixed-length shift-register, as the FIFO data-ports will only be w bits wide, rather than the total of $2w$ supported by the underlying RAM. Another drawback is that in some hardened FIFO circuitry, it is not possible to have the RAM start-up with a fixed number of data elements already in the FIFO (or at least this functionality is not supported by the toolchain), so the logic saved in implementing the FIFO counters is offset by the state machine needed to initialised the RNG.

If hardened FIFOs are not used, there are a number of ways in which RAMs and user logic can be used to construct FIFOs, with the possible choices limited by the features of the RAM. The three main RAM characteristics are: the width w, and number of elements 2^a; the number of ports, i.e. how many addresses can be accessed per cycle; and per-port Read-Before-Write (RBW) support, which is whether the old data at the port's address can be read while the new data is being written. We identify five types of FIFO that can be implemented on a RAM with given characteristics (show graphically in Figure 1):

1. Single RBW port: single $w \times k$ FIFO, $k \leq 2^a$.
2. Dual port, one Write-Only, one Read-Only: single $w \times k$ FIFO, $k < 2^a$.
3. Dual port, one RBW, one Read-Only: single $w \times k$ FIFO, $k \leq 2^a$, with a tap at an arbitrary offset $1 < l < k$.
4. Dual port, both RBW: single $2w \times k$ FIFO, where $k \leq 2^a$.
5. Dual port, both RBW: two independent FIFOs of size $w \times k_1$ and $w_\times k_2$ where $k_1 + k_2 \leq 2^a$.

Option 5 is a generalisation of option 4, which trades the ability to have different length FIFOs for the increased logic needed to implemented two counters. However, option 4 is functionally equivalent to option 1, so from a functional point of view, these resources offer just two kinds of (fixed-length) FIFO: basic FIFOs, and FIFOs with a single tap. Underlying architectural details will limit the possible configurations, but, along with the basic LE, these are the only relevant functional primitives available.

In order to maximise the memory bandwidth for current FPGA architectures, this means we need to be able to accommodate RNGs built using LEs and one of three combinations of fixed-length FIFOs:

- A single FIFO.
- A single FIFO with one tap.
- A pair of FIFOs, with a common width, but different lengths.

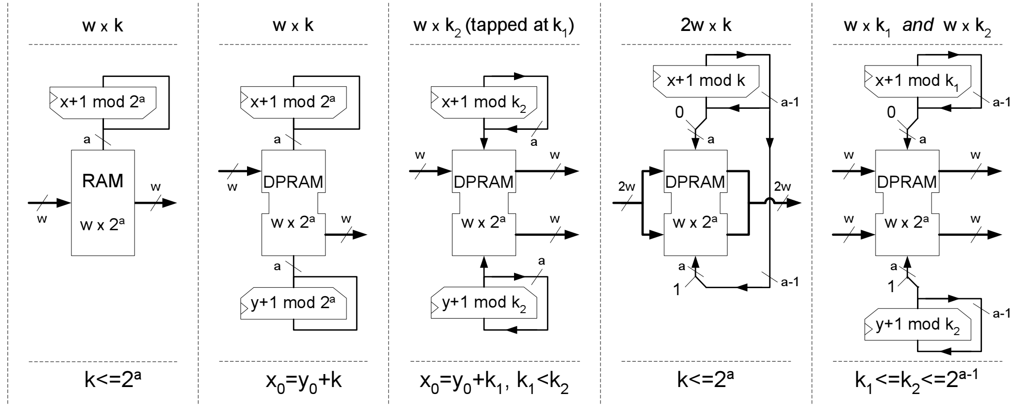

Figure 1: BRAM FIFO modes

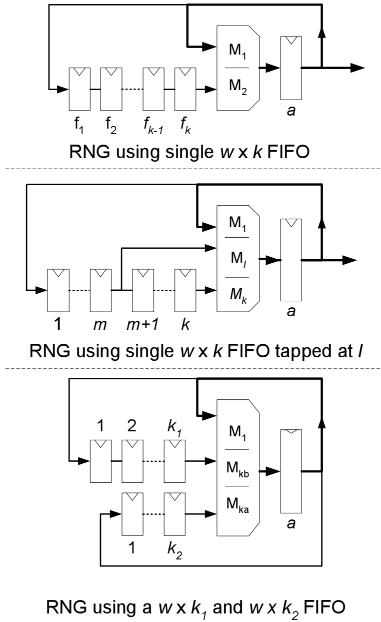

Figure 2: RNG feedback modes

A diagram of each RNG is shown in Figure 2, but while it is simple to draw a high-level diagram, it is much more difficult to choose the actual parameters. One cannot simply throw together collections of FIFOs and exclusive-or gates and hope to meet all seven requirements from the previous section. Each RNG must be carefully designed, to guarantee that hard guarantees such as performance and efficiency are always achieved, and that the necessary conditions for more complex quality and period constraints are maintained.

4.2 Single $w \times k$ word-based FIFO

Given a $w \times k$ bit FIFO, and a requirement for r random bits per cycle, we wish to create a minimal-resource recurrence containing $n = wk + r$ bits. Initially, we will assume that $r = w$, so $n = w(k+1)$, then later treat the general (and more important) case where w is not a factor of r. First split the state s into $k+1$ groups of w bits, $\mathbf{s}_i = (\mathbf{a}_i, \mathbf{f}_{i,1}, ..., \mathbf{f}_{i,k})$. We will use the convention that $\mathbf{a}_i$ contains the active bits from state $\mathbf{s}_i$, and $\mathbf{f}_{i,1}, \mathbf{f}_{i,2}, ...$ represent the FIFO bits of the state (where there is more than one FIFO $\mathbf{g}$ will be used for the second FIFO).

The $n \times n$ recurrence matrix can then be defined using a $(k+1) \times (k+1)$ grid of $w \times w$ matrices:

$$\mathbf{A} = \begin{pmatrix} \mathbf{M}_1 & 0 & \dots & 0 & \mathbf{M}_k \\ \mathbf{I}_w & 0 & \dots & 0 & 0 \\ 0 & \mathbf{I}_w & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & \mathbf{I}_w & 0 \end{pmatrix} \tag{6}$$

where $\mathbf{I}_w$ is the $w \times w$ identity matrix, and $\mathbf{M}_1$ and $\mathbf{M}_k$ are two arbitrary $w \times w$ matrices. The mapping is more easily understood as a relation between the next and current states:

$$\begin{pmatrix} \mathbf{a}_{i+1} \\ \mathbf{f}_{i+1,1} \\ \mathbf{f}_{i+1,2} \\ \vdots \\ \mathbf{f}_{i+1,k} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_1\mathbf{a}_i \oplus \mathbf{M}_2\mathbf{f}_{i,k} \\ \mathbf{a}_i \\ \mathbf{f}_{i,1} \\ \vdots \\ \mathbf{f}_{i,k-1} \end{pmatrix} \tag{7}$$

There are a number of requirements on the two matrices $\mathbf{M}_1$ and $\mathbf{M}_k$. A necessary criteria for maximum period is that each active bit from $\mathbf{a}_i$ must be used at least once, and similarly each bit from the FIFO output $\mathbf{f}_{i,k}$ must be used at least once. This condition is met if each column of the two matrices contains at least one non-zero element. Note that this is a necessary but not sufficient condition for maximum period; a given $(w, k, \mathbf{M}_1, \mathbf{M}_k)$ recurrence also needs to be tested for primitivity before we know if the sequence will have period $2^{w(k+1)} - 1$ (see Section 5).

To meet our requirements on resource usage and critical path, there are also restrictions on the number of ones in each row of the $w \times 2w$ matrix $\mathbf{M} = [\mathbf{M}_1\ \mathbf{M}_k]$. Each row represents the inputs to an active bit, so there must be at least two ones in each row of $\mathbf{M}$ (as otherwise it would be a FIFO bit). However, each FPGA family has a maximum number of inputs that can be supported per LUT, so to avoid the use of two LUTs per active bit, the number of ones in each row must not exceed the number of LUT inputs.

4.3 Supporting $w \neq r$

The recurrence just discussed assumes that $r = w$, so the number of random bits required is the same as the FIFO width, but this is not usually the case, for two reasons. The first is simply that a given application might need more or

RAM Resource		Prime	Wasted	Prime	Excess
Aspect	Bits	below	RAM bits	above	LE bits
36×128	4608	4423	4%	9689	2081
36×256	9216	4423	52%	9689	473
36×512	18432	11213	39%	19937	1505

Table 2: Mersenne primes bracketing typical block RAM sizes.

less than w bits per cycle. In the case that the application requires less than w bits we argue that the generator itself should still use $r = w$, as this maximises the utilisation of memory bandwidth, and maximises the change in RNG state per-cycle. However, if the number of required random bits is greater than w then clearly the RNG must support a larger number of active bits.

The second reason is that testing whether a recurrence of length n produces a maximum length sequence becomes computationally infeasible if n is much larger than 1500, as it is necessary to know all factors of $2^n - 1$. A 36×512 RAM can hold approximately 18000 bits, and we would like to use as many as possible, but it is effectively impossible to determine all factors of an arbitrary integer $2^n - 1$ when n is close to 18000. However, if we pick n such that $2^n - 1$ is a Mersenne prime, then we know that there are no factors, so the test for primitivity reduces to a much cheaper test for irreducibility. A necessary condition for $2^n - 1$ being a Mersenne prime is that n itself is prime, hence w will never be a factor of n.

The Mersenne primes are of interest in many fields, and have been exhaustively documented for $n \leq 13466917$. Unfortunately, there are not many Mersenne primes, with only 14 occurring in the range $10^3 < n < 10^5$. Given a FIFO that can contain some number of bits, one can either choose the least Mersenne prime above that size, and implement the excess bits in logic, or choose the greatest Mersenne prime below that size, and waste some of the FIFO's storage capacity. Table 2 summarises these bounds above and below for typical block RAM sizes.

In all three cases the upper prime requires rather a large number of excess bits, so to meet our requirement for logic efficiency the lower bound must be used. However, the maximum use of RAM storage is not part of our requirements, only that the maximum RAM bandwidth is used: a RAM component takes up the same area no matter how much of it is used, so as long as the period is $\geq 2^{2000}$, the exact number of RAM bits used is of less interest than maximising the transformation rate of the bits in the RAM.

To implement a generator with $r > w$, the matrix $\mathbf{A}$ must be partitioned into a non-regular grid of sub-matrices. The first column contains sub-matrices with width r, while the following columns have width w, and similarly the first row has height r while the rest have height w (the widths and heights are shown at the left and top):

$$\begin{array}{c|cccccc} & r & w & \dots & w & w \\ \hline r & \mathbf{M}_1 & 0 & \dots & 0 & \mathbf{M}_k \\ w & \mathbf{S} & 0 & \dots & 0 & 0 \\ w & 0 & \mathbf{I}_w & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ w & 0 & 0 & \dots & \mathbf{I}_w & 0 \end{array} \quad (8)$$

where $\mathbf{M}_1$ is a $r \times r$ matrix, $\mathbf{M}_k$ is a $r \times w$ matrix, with the same conditions on the rows and columns of the composite matrix $\mathbf{M}$ as before.

The new $w \times r$ matrix $\mathbf{S}$ selects w bits from the r active bits to be fed into the FIFO. There must be exactly one non-zero element in each row of $\mathbf{S}$, as the FIFO input cannot implement logic to combine two or more values, and if there are no ones then nothing enters the FIFO bit represented by that column. Additionally, the number of ones in each column of $\mathbf{S}$ can be *at most* one, as if two ones occurred, the same data would be present in two bits of the FIFO, reducing the effective width to $w - 1$. A simple solution is to choose $\mathbf{S} = [\mathbf{I}_w \ 0]$.

4.4 Single $w \times k$ FIFO tapped at l

A FIFO with a tap offers the opportunity to decrease the lag k auto-correlations, by also combining values from lag l. This requires the addition of another $r \times k$ matrix $\mathbf{M}_l$ to the recurrence:

$$\begin{pmatrix} \mathbf{a}_{i+1} \\ \mathbf{f}_{i+1,1} \\ \mathbf{f}_{i+1,2} \\ \vdots \\ \mathbf{f}_{i+1,l} \\ \vdots \\ \mathbf{f}_{i+1,k} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_1\mathbf{a}_i \oplus \mathbf{M}_l\mathbf{f}_{i,l} \oplus \mathbf{M}_k\mathbf{f}_{i,k} \\ \mathbf{S}\mathbf{a}_i \\ \mathbf{f}_{i,1} \\ \vdots \\ \mathbf{f}_{i,l-1} \\ \vdots \\ \mathbf{f}_{i,k-1} \end{pmatrix} \quad (9)$$

The previous restrictions on the numbers of ones in each row must now be extended to the matrix $\mathbf{M} = [\mathbf{M}_1 \ \mathbf{M}_l \ \mathbf{M}_k]$. $\mathbf{M}_k$ must clearly still have one or more ones in each column to provide a maximum period, as must $\mathbf{M}_l$ to meet the requirement to maximise FIFO bandwidth. However, if $\gcd(t, k) = 1$ and $r = w$ it is not always necessary for $\mathbf{M_1}$ to contain any ones to achieve maximum period (in other cases it should).

This arrangement of a recurrence with three lags corresponds closely to the design of many software generators, as it spreads out the auto-correlations in the output stream, but requires only two memory reads per generated word.

4.5 Two different length FIFOs

As shown earlier, a $w \times 2^a$ RAM with two Read-Before-Write ports can be used to implement either one $2w \times k$ FIFO (where $k \leq 2^a$), or two FIFOs of length $w \times k_1$ and $w \times k_2$ (where $k_1 + k_2 \leq 2^a$). Both arrangements provide maximum possible bandwidth, but implementing two FIFOs requires slightly more logic resources (as two address generators are needed). However, this disadvantage is outweighed by the ability to reduce the potential auto-correlations caused when the active bits $\mathbf{a}_{i+1}$ are formed only from $\mathbf{a}_i$ and $\mathbf{a}_{i-k}$.

Another advantage is that, because w is fixed by the architecture, r by the application, and n by the closest Mersenne prime, the value of k in a $2w \times k$ FIFO is essentially fixed. However, with two FIFOs we can choose any lengths satisfying $k = k_1 + k_2$, allowing the maximum lag to be longer.

The direct recurrence form of a two FIFO generator is:

$$\begin{pmatrix} \mathbf{a}_{i+1} \\ \mathbf{f}_{i+1,1} \\ \mathbf{f}_{i+1,2} \\ \vdots \\ \mathbf{f}_{i+1,k_1} \\ \mathbf{g}_{i+1,1} \\ \mathbf{g}_{i+1,2} \\ \vdots \\ \mathbf{g}_{i+1,k_2} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_1\mathbf{a}_i \oplus \mathbf{M}_{k_1}\mathbf{f}_{k_1} \oplus \mathbf{M}_{k_2}\mathbf{g}_{k_2} \\ \mathbf{S}_f\mathbf{a}_i \\ \mathbf{f}_{i,1} \\ \vdots \\ \mathbf{f}_{i,k_1-1} \\ \mathbf{S}_g\mathbf{a}_i \\ \mathbf{g}_{i,1} \\ \vdots \\ \mathbf{g}_{i,k_2-1} \end{pmatrix} \quad (10)$$

Family	RAM	w	2^a	Mode	Bits	n	FIFOs	r
Virtex-E	RAMB4	16	256	2xRAW	4096	3217	16 x 199	33
Stratix-II	M4K	18	256	2xRBW	4608	4423	18x158, 18x85	49
Stratix-III	M9K	18	512	2xRBW	9216	4423	18x158, 18x85	49
Virtex-4	RAMB16	36	512	2xRBW	18432	11213	36x202, 36x107	89
Virtex-5	RAMB36	36	1024	2xRBW	36864	23209	36x425, 36x217	97

Table 3: Parametrisations of RNG for specific architectures.

with the following matrix sizes: $\mathbf{M}_1 : r \times r$; $\mathbf{M}_{k_1}, \mathbf{M}_{k_2} : w \times r$; and $\mathbf{S}_f, \mathbf{S}_g : r \times w$. To meet our requirements the following conditions must be met: each row of $\mathbf{M} = [\mathbf{M}_1\ \mathbf{M}_{k_1}\ \mathbf{M}_{k_2}]$ must be implementable using one LUT; each column of $\mathbf{M}$ must contain one or more ones; each column of $\mathbf{S} = [\mathbf{S}_f\ \mathbf{S}_g]$ must contain a single one; and each row of $\mathbf{S}$ must contain at most a single one.

4.6 Practical Parametrisations

The previous discussion was, by intention, architecture independent, but it is useful to examine how real world architectures fit within this framework. Table 3 shows five device families and a summary of the block RAM found within each. There are many potential choices, such as the number of output bits, and the ratio between FIFO sizes, but the table shows a sensible default. The maximum Mersenne Prime that fits in the RAM is chosen, then the minimum number of active bits greater than or equal to the number of FIFO inputs is chosen. The FIFO lengths are chosen by looking for relatively prime pairs where one length is roughly double the other.

The families of recurrences presented in this section ensure that we meet some of our requirements: performance (by making sure that the critical path is FF-LUT-FF), maximum memory bandwidth utilisation, and logic efficiency (as each active bit uses exactly one LUT). However, they only provide *necessary* conditions for maximum period, quality, and polynomial density, they don't guarantee it. In the next section we explain how one can search for specific generators that have these properties.

5. FINDING MAXIMUM LENGTH RECURRENCES

There is no way of directly constructing a maximum-length linear recurrence, instead we must define a parameter space that provides necessary (but not sufficient) conditions, then search within that space until a maximum-length sequence is found. The previous section identified a number of parameter spaces, but they are clearly too large to search exhaustively. Instead we must choose and test random points within the space, hoping that eventually one of the parametrisations will produce a maximum length sequence.

5.1 Selecting Random Parameters

Given one of the parameter-spaces from the previous section, it is relatively easy to generate a random point (i.e. RNG parametrisation) within the space. For example, to generate a $r \times (r + w)$ matrix $\mathbf{S}$ with t ones in each row and at least one one in each column, start with the matrix $[\mathbf{I}_r\ 0_{r\times w}]$, and randomly permute the columns. Then randomly select elements within the matrix, and if the count in that element's row is less than t, set the element to one.

Our only advice on the random generation of recurrences is that one should not attempt to find recurrences where every single active bit has the same number of LUT inputs. For some parameter spaces it is impossible to find a maximum length generator of this type, and in general it is better to have a mix of active bit arities; for example, in our search process for 4-LUT architectures we choose approximately 50% to have four inputs, and the rest to have three.

5.2 Checking for Maximum Period

Given a fully parametrised linear recurrence, we must check that the characteristic polynomial is primitive. In the previous work on LUT optimised generators [13], this was achieved by first building the explicit recurrence matrix, then calculating the characteristic polynomial from the matrix. However, for $n > 1000$ the calculation of the characteristic polynomial starts to dominate the search process, and becomes completely impractical for $n > 2000$. As we are interested in Mersenne primes such as $n = 11213$, this method is infeasible.

In the approach suggested here the explicit recurrence matrix is never needed; instead, we rely on the fact that the characteristic polynomial directly determines the bit sequence of each RNG output (see Equation 3). First run the generator for $2n$ steps (from any state), and collect a sequence of $2n$ successive values from one bit of the output. The Berlekamp-Massey algorithm [8] is then able to determine the minimal polynomial for generating that sequence (the characteristic polynomial) in $O(n^2)$ time.

The process can be described as follows (where appropriate we have indicated the appropriate function in NTL [10], an efficient library for calculations in $\mathbb{F}_2$):

1. Generate a random candidate RNG.
2. Choose an initial (non-zero) state $\mathbf{s}_0$ and calculate the next $2n$ RNG outputs, $\mathbf{x}_1..\mathbf{x}_{2n}$.
3. Choose a bit offset within the output (any will do), and form the bit vector $\mathbf{b} = b_1, .., b_{2n}$ from $\mathbf{x}_1..\mathbf{x}_{2n}$.
4. Calculate P_z, the minimum generating polynomial of $\mathbf{b}$, using the Berlekamp-Massey algorithm. For example: `Pz=NTL::MinPolySeq(b,n);`.
5. Check the degree of P_z. If it is less than n then the RNG is not maximum period, so go back to Step 1.
6. Check that P_z is irreducible. If it is reducible then the RNG is not maximum period, so go to back Step 1. If $2^n - 1$ is a Mersenne prime then the RNG has maximum period, so exit the search. Irreducibility can be tested using `isIrred=NTL::IterIrredTest(Pz);`
7. Perform a full primitivity test on P_z. If it is primitive, then exit the search, otherwise go back to Step 1. The

Family	MHz	RAM	LUT	FF	Slices
Spartan-3	*245*	1	115	181	110
Virtex-II Pro	*385*	1	115	181	117
Virtex-4	*511*	1	117	109	73
Virtex-5	*624*	1	113	109	36×2

Table 4: Area and clock-rate (post-PAR static-analysis) for reference generator with $n = 11213$ and $r = 89$.

program PPSearch [1] is able to perform this computation, but only if a factorisation of $2^n - 1$ is available.

The search process typically takes a few hours on a contemporary single-processor machine (e.g. Pentium-4 3GHz), depending on the complexity of the parameter set. Increases in n and r tend to increase the search time, but so far we have not found any sets of parameters with $n \leq 23209$ and $r \leq 521$ where no maximum period generator could be found in ~ 10 CPU days. The search process is also inherently parallelisable, and can be trivially scaled over hundreds of machines using a grid service such as Condor [1]. Using the ~ 400 machine Condor pool at Imperial College Department of Computing, we find that a maximum period generator can usually be found within 15 minutes.

6. EVALUATION

The results for a reference parametrisation are shown in Table 4 for a number of different FPGA families. We picked the combination of 36×512 dual ported RBW RAMs and 4 input LUTs, as this broadly matches the Xilinx family of architectures, and allows a period of $2^{11213} - 1$. The RAM is split into two FIFOs of size 36×202 and 36×107 ($\gcd(202, 107) = 1$), resulting in 11052 FIFO bits, and 89 active bits.

The same VHDL description is used for all generators, with the RAMs inferred from the VHDL. No timing constraints, placement constraints or optimisation hints are used, and default tool-chain settings and effort levels are used, except for timing driven placement, which is enabled. All have been verified in simulation, and the Virtex-2 and Virtex-4 generators have also been tested in hardware. Note that the reported clock rates are those reported by the toolchain for post place-and-route static-timing: we have not tested the operation of hardware at these speeds, and give them only as an indication of the delay due to routing and components.

In every family with RAM output registers, the limiting performance is found to be the speed of the RAM, not logic or routing. In families without output registers, the speed is limited by the routing from RAM outputs into LE FFs, but the observed speeds are not significantly different than those observed in an artificial I->FF->RAM->FF->O circuit.

The resource usage is also close to the minimum possible, consisting of just the r LUTs required for the active bits, and a small number for address generation. The increase in FFs for some families is due to unregistered RAM outputs, but this is unavoidable in such architectures if the maximum clock speed is to be achieved.

Table 5 explores different parametrisations in the Virtex-5 architectures. In each case a clock-constraint of 550MHz is applied, the maximum speed of the block RAMs, and it was achieved in each case. Group (a) shows the case where more than four inputs are used per active bit, using the Virtex-5 platform which has 6-LUTs. Increasing the number of inputs per active-bit does not appear to decrease the clock rate, thanks to the single LUT critical path. We strongly suspect that increasing the number of inputs increases the quality (this is supported by the difference in quality between 3- and 4-input generators in [13]), but the basic quality of the generators is so high that none of the empirical tests can observe any difference (see notes on empirical tests below).

In group (b) the number of active bits is increased, so each RNG provides more bits per cycles. Note that each design still achieved 550MHz without any noticeable increase in the time taken to place-and-route each design, supporting our claim that the generator is able to easily operate at the maximum possible speed, without any changes in compiler settings, device specific HDL, or placement constraints. We would observe that these results are derived from a design containing only the RNG, so in a real-world design the RNG clock-rate might deteriorate when generating so many bits. However, in such a design the performance of other logic is likely to degrade at a faster rate, so the RNG should never form a performance bottle-neck.

Group (c) looks at using smaller FIFO resources, in this case the 1×33 SRL32 shift-register found in the Virtex-5 architecture. The first generator uses 8 SRL32s as a 8×33 FIFO, and in only 16 LEs can provide 8 bits per cycle with a period of $2^{268} - 1$. The second generator uses 32 SRL32s, grouped into 16×33 and 16×32 FIFOs, producing 32 bits per cycle in only 64 LEs. These generators do *not* meet our set of seven requirements, as the periods are both too short, but they may be of use in applications with lesser requirements.

The SRL32 generators are also faster than the RAM based generators: static-timing reports that both run in excess of 750MHz, faster than the maximum speed of the global clock net (710MHz). Such high clock-rates are too high to be taken seriously without extensive testing in a real device; however, such generators may be useful when high quality random bits are needed close to IOs, particularly as the small number of resources would minimise the fanout of high speed clocks.

Finally, group (d) shows an example with a much longer period, achieved by using two block RAMs. This demonstrates that longer period generators are possible and remain efficient, but in practise the period 11213 generators probably provide a sufficient period for most applications and only require one block RAM.

In terms of quality, all the generators have been tested using the Diehard battery [7], and the following batteries from TestU01 [5]: SmallCrush, Crush, BigCrush, and Alphabit (using 2^{40} bits). Each battery is run twice for each generator, and a test is considered passed if both runs produce p-values in the range $[10^{-6}..1-10^{-6}]$ and one of the p-values is inside the range [0.01..0.99]. Because so many tests are applied during this evaluation process, a small number are expected to produce p-value pairs that do not meet the second criteria. These are resolved by repeating the test an additional two times, and verifying that an Anderson-Darling test over the four p-values produces a significance outside the range $[10^{-6}..1 - 10^{-6}]$. The tests all require 32 bit samples, so for each generator with $r \geq 32$ the first 32 active bits are used, while for the 8-bit generator successive groups of outputs are used to form 32-bit samples. The reference generator from group (a) is additionally tested using 16 dif-

	n log_2 period	w FIFO width	t LUT inputs	r Bits per cycle	$w(P_z)$ Poly weight	RAM	LUT	FF	LUT-FF pairs (efficiency)	Slices
(a)	11213	36	6	89	0.493	1	113	109	113 (79%)	36
(b)	11213	36	4	161	0.500	1	184	181	186 (87%)	60
	11213	36	4	269	0.503	1	291	289	294 (91%)	88
	11213	36	4	341	0.501	1	364	361	365 (93%)	102
	11213	36	4	521	0.498	1	544	541	545 (96%)	158
(c)	268	4	5	8	0.131	0	16	16	16 (50%)	6
	1072	16	6	32	0.404	0	64	64	64 (50%)	20
(d)	23209	72	6	169	0.497	2	193	189	195 (87%)	61

Table 5: Summary of area and performance results for different RNG parametrisations.

ferent random 32-bit subsets from the 89 active bits to check that the bits are all of equal quality.

The only tests that are not passed are those that look for linear dependencies, specifically the Matrix-Rank and Linear-Complexity tests [5]. The tests are passed for short-range dependencies, but once dependencies spanning more than n successive random numbers are considered the tests are failed. However, these tests are impossible for any binary linear recurrence to pass, and are mainly of interest for cryptographic applications: even MT19937 does not pass these tests, and it has been successfully used in Monte-Carlo simulations for many years. Every non-cryptographic quality family of generators has known systemic flaws, and these tests should not be seen as reflecting badly on the quality of the generators when used for non-cryptographic applications.

Except for the tests for linear dependencies, the batteries are all convincingly passed, and no observable difference in quality between the generators is seen. We also repeat the batteries 8 times for the reference generator, then perform an Anderson-Darling test on each set of 8 p-values, and no p-values outside the range $[10^{-6}..1 - 10^{-6}]$ are observed.

7. COMPARISON TO RELATED WORK

Table 6 provides a summary of different methods for random number generation in FPGAs. This table ignores clock rates, and looks purely at resources: this actually biases the comparison against the LUT-FIFO generators introduced in this paper, as they all achieve the maximum possible rate, but it makes for a simpler comparison. The Virtex-II Pro platform is used, as the most recent paper, on the TT800 [11] used this platform. All generators presented from [13] have been re-placed and routed for this platform, and the LFSR is generated using Xilinx Core-Gen.

We have included the results from the two fastest bandwidth generators in [11] and [3], though both have sacrificed guaranteed period for performance. In both cases the authors have extended the software generators on which they are based, to create generators that produce more than one 32-bit word per cycle. However, the way in which they are extended does not produce the same sequence as the original software, and does not necessarily have maximum period length. So while the tested empirical quality still appears good, the generators run the risk of entering low period cycles when initialised from certain seeds. These generators are shown with a question mark in the period column of the table.

It is difficult to compare the quality of random number generators, as the empirical tests are not able to provide fine levels of distinction. We use the arbitrary categories of - poor: fails Diehard; medium: passes Diehard but fails some tests in Crush that don't look for linear dependencies; and good: passes all known tests except those for linear dependencies. All the LUT-FIFO generators fall into the good category, meeting our goal of empirical quality.

The next column shows the weight of the characteristic polynomial, both as the total number of ones, and as a fraction of the polynomial degree. The LUT-FIFO generators have by far the highest absolute polynomial weight, and also have the most balanced polynomials (fraction of ones closest to 0.5). Note that the well-regarded MT19937 only has 135 terms, so by comparison the LUT-FF generators with > 5000 terms will provide far less detectable autocorrelations in stream state.

One of the most interesting columns in the table is the last, the number of random bits generated per LUT, which can seen as a measure of efficiency. The LUT optimised generator is impossible to beat, as it uses exactly one LUT per bit. However, the generators presented can come close to this level, with only a small reduction in efficiency due to the FIFO counting logic, and have the great advantage that they provide a sequence over 2^{10000} longer than the LUT optimised generator. All the other generators have much lower efficiency.

8. CONCLUSION

In this paper we have presented a methodology for designing high-quality long-period Random Number Generators, taking into account the requirements and resources of FPGAs. Two basic FPGA primitives, LUT-FF pairs and FIFOs, are used to directly construct bit-oriented RNGs, rather than adapting RNGs originally designed for the word-level instructions found in software. The suggested approach has a number of advantages:

1. High clock-rates are guaranteed by making sure the longest path is always FF-LUT-FF or FF-RAM-FF.
2. The generators can be very efficient in terms of LE utilisation per random bit, achieving close to the practical maximum efficiency.
3. Very high periods can be achieved using a single block-RAM, while generating large numbers of bits per cycle.
4. The RNGs can be customised for the needs of the application, with a single generator producing as many bits per cycle as the application requires.

Generator	Quality	n	r	$w(P_z)$		BRAM	LUT	FF	Slices	r/LUT
		log_2(period)	Bits/cycle	Poly weight						
LFSR-160 [13]	Poor	160	32	5	(0.031)	0	448	384	256	0.07
Linrec (t=3) [13]	Medium	512	512	235	(0.459)	0	513	512	292	1.00
Taus113 [4]	Medium	113	32	49	(0.434)	0	87	208	143	0.37
TT800 (1 port) [11]	Medium	800	32	261	(0.326)	2	162	162	81	0.26
TT800 (24 port) [11]	Medium	?	768	?	?	0	5060	5060	2530	0.15
LUT-FIFO (RAM,t=4)	Good	11213	89	5299	(0.473)	1	115	181	133	0.77
LUT-FIFO (RAM,t=4)	Good	11213	521	5585	(0.498)	1	539	611	403	0.97
MT19937 (PMT) [3]	Good	19937	32	135	(0.007)	2	278	-	-	0.12
MT19937 (FMT52) [3]	Good	?	1664	?	?	0	22926	-	-	0.07

Table 6: Comparison of different generators in the Virtex-II Pro platform.

5. Each output bit is equally random, rather than concentrating on the quality of the MSBs, so the bits can be arbitrarily regrouped into integers.
6. The amount of RNG state modified per cycle is maximised by using all available bandwidth per block RAM.

RNGs developed using this methodology have been implemented in a number of different FPGA families, and in all cases a clock rate limited only by the underlying RAM and near optimal LE utilisation is seen. In terms of Gb achieved per LUT, the RNGs are the most efficient long period random number generators, and, except for the flaws common to all binary linear recurrences, the generated streams have no statistical defects.

One avenue of future work is to explore the parameter space for this kind of generator, and identify heuristics that improve both the speed of the search process, and the quality of the generated stream. Part of this would involve looking at the equidistribution of generated streams, a measure of statistical quality over the entire RNG period. Another avenue of exploration is to operate over higher order finite fields.

9. ACKNOWLEDGEMENTS

The support of UK Engineering and Physical Sciences Research Council (Grant references. EP/D062322/1 and EP/C549481/1), Celoxica and Xilinx is gratefully acknowledged.

10. REFERENCES

[1] S. Duplichan. PPSearch : A primitive polynomial search program. http://users2.ev1.net/~sduplichan/primitivepolynomials/, 2003.

[2] P. Kohlbrenner and K. Gaj. An embedded true random number generator for FPGAs. In *Proc. ACM/SIGDA Int. Symposium on Field-Programmable Gate Arrays*, pages 71–78, 2004.

[3] S. Konuma and S. Ichikawa. Design and evaluation of hardware pseudo-random number generator mt19937. *IEICE - Trans. Inf. Syst.*, E88-D(12):2876–2879, 2005.

[4] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269, 1999.

[5] P. L'Ecuyer and R. Simard. TestU01 random number test suite. www.iro.umontreal.ca/~simardr/indexe.html, 2007.

[6] D. Lee, J. Villasenor, W. Luk, and P. Leong. A hardware Gaussian noise generator using the box-muller method and its error analysis. *IEEE Transactions on Computers*, 55(6):659–671, 2006.

[7] G. Marsaglia. The Diehard random number test suite. http://stat.fsu.edu/pub/diehard/, 1997.

[8] J. Massey. Shift-register synthesis and bch decoding. *IEEE Trans. on Information Theory*, 15(1):122–127, 1969.

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

[10] V. Shoup. Ntl: A library for doing number theory. http://www.shoup.net/ntl/.

[11] V. Sriram and D. Kearney. A high throughput area time efficient pseudo uniform random number generator based on the TT800 algorithm. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 529–532, 2007.

[12] D. B. Thomas and W. Luk. Efficient hardware generation of random variates with arbitrary distributions. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 57–66, 2006.

[13] D. B. Thomas and W. Luk. High quality uniform random number generation using LUT optimised state-transition matrices. *Journal of VLSI Signal Processing*, 47(1), 2007.

[14] D. B. Thomas and W. Luk. Sampling from the multivariate Gaussian distribution using reconfigurable hardware. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 3–12, 2007.

[15] G. L. Zhang, P. H. Leong, D.-U. Lee, J. D. Villasenor, R. C. Cheung, and W. Luk. Ziggurat-based hardware Gaussian random number generator. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 275–280. IEEE Computer Society Press, 2005.