The LUT-SR Family of Uniform Random Number Generators for FPGA Architectures

David B. Thomas, *Member, IEEE,* and Wayne Luk, *Member, IEEE*

***Abstract*—FPGA-optimised Random Number Generators (RNGs) are more resource efficient than software-optimised RNGs, as they can take advantage of bit-wise operations and FPGA-specific features. However, it is difficult to concisely describe FPGA-optimised RNGs, so they are not commonly used in real-world designs. This paper describes a type of FPGA RNG called a LUT-SR RNG, which takes advantage of bit-wise XOR operations and the ability to turn LUTs into shift-registers of varying lengths. This provides a good resource-quality balance compared to previous FPGA-optimised generators, between the previous high-resource high-period LUT-FIFO RNGs and low-resource low-quality LUT-OPT RNGs, with quality comparable to the best software generators. The LUT-SR generators can also be expressed using a simple C++ algorithm contained within the paper, allowing 60 fully-specified LUT-SR RNGs with different characteristics to be embedded in the paper, backed up by an online set of VHDL generators and test-benches.**

***Index Terms*—FPGA, Uniform Random Number Generator, Equidistribution**

I. Introduction

MONTE Carlo applications are ideally suited to FPGAs, due to the highly parallel nature of the applications, and because it is possible to take advantage of hardware features to create very efficient random number generators. In particular, uniform random bits are extremely cheap to generate in an FPGA, as large numbers of bits can be generated per cycle at high clock-rates using LUT-OPT [1] or LUT-FIFO generators [2]. In addition, these generators can be customised to meet the exact requirements of the application, both in terms of the number of bits required per cycle, and for the FPGA architecture of the target platform.

Despite these advantages, FPGA-optimised generators are not widely used in practise, as the process of constructing a generator for a given parametrisation is time-consuming, in terms of both developer man-hours and CPU-time. While it is possible to construct all possible generators ahead of time, the resulting set of cores would require many megabytes, and be difficult to integrate into existing tools and design-flows. Faced with these unpalatable choices, engineers under time constraints understandably choose less efficient methods, such as Combined Tausworthe generators [3], or parallel LFSRs.

This paper describes a family of generators which makes it easier to use FPGA-optimised generators, by providing a simple method for engineers to instantiate an RNG which meets the specific needs of their application. Specifically, it shows how to create a family of generators called LUT-SR RNGs, which use LUTs as shift-registers to achieve high-quality and long periods, while requiring very few resources. The main contributions are:

- A type of FPGA-optimised uniform RNG called a LUT-SR generator, which uses LUT-based shift-registers to implement generators with periods of $2^{1024}-1$ or more, using two LUTs and two FFs per generated random bit.
- An algorithm for describing LUT-SR RNGs using five integers, and a set of open-source test-benches and tools.
- Tables of 60 LUT-SR RNGs covering output widths from 32 up to 624, with periods from $2^{1024}-1$ up to $2^{19937}-1$.
- A theoretical quality analysis of the given RNGs in terms of equidistribution, and a comparison with other software and hardware RNGs.

The LUT-SR family was first presented in a conference paper [4], which concentrated on the practical aspects of constructing and using these generators. This paper adds Section V which describes the method used to find maximum period generators, Section VI describing the process used to select the highest quality generators, and a rigorous theoretical quality analysis in terms of equidistribution in Section VIII.

II. Overview of Binary Linear RNGs

The LUT-SR RNGs are part of a large family of RNGs, all of which are based on binary linear recurrences. This family includes many of the most popular contemporary software generators, such as the Mersenne Twister (MT-19937) [5], the Combined Tausworthe (TAUS-113) [3], SFMT [6], WELL [7], and TT-800 [8]. This section gives an overview of the underlying maths, and describes existing binary linear RNGs used in FPGAs.

A. Binary Linear RNGs

Binary linear recurrences operate on bits (binary digits), where addition and multiplication of bits is implemented using exclusive-or ($\oplus$) and bitwise-and ($\otimes$). [1] The recurrence of an RNG with n-bit state and r-bit outputs is defined as:

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i \tag{1}$$

$$\mathbf{y}_{i+1} = \mathbf{B}\mathbf{x}_{i+1} \tag{2}$$

where $\mathbf{x}_i = (x_{i,1}, x_{i,2}, ..., x_{i,n})^\mathsf{T}$ is the n-bit state of the generator; $\mathbf{y}_i = (y_{i,1}, y_{i,2}, ..., y_{i,r})^\mathsf{T}$ is the r-bit output of the

The support of UK Engineering and Physical Sciences Research Council (Grants EP/D062322/1 and EP/C549481/1), Alpha Data, and Xilinx is gratefully acknowledged.

D. Thomas is with the Dept. of Electrical and Electronic Engineering, Imperial College London (dt10@imperial.ac.uk).

W. Luk is with the Dept. of Computing, Imperial College London (wl@doc.ic.ac.uk).

[1] This is the same as operating modulo-2, or in the Galois field GF(2).

generator; $\mathbf{A}$ is an $n \times n$ binary transition matrix; and $\mathbf{B}$ is an $r \times n$ binary output matrix. Because the state is finite, and the recurrence is deterministic, eventually the state sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ...$ must start to repeat.

The minimum value p such that $\mathbf{x}_{i+p} = \mathbf{x}_i$ is called the period of the generator, and one goal in designing random number generators is to achieve the maximum period of $p = 2^n - 1$; a period of 2^n cannot be achieved because it is impossible to choose $\mathbf{A}$ such that $\mathbf{x}_0 = \mathbf{0}$ maps to anything other than $\mathbf{x}_1 = \mathbf{0}$. This leads to two sequences in a maximum period generator: a degenerate sequence of length one which contains only zero, and the main sequence which iterates through every possible non-zero n-bit pattern before repeating. A necessary and sufficient condition for a generator to have maximum period is that the characteristic polynomial $P(z)$ of the transition matrix $\mathbf{A}$ must be primitive [1].

The matrix $\mathbf{B}$ is used to transform the internal RNG state into the random output bits produced by the generator. In the simplest case we have $r = n$ and $\mathbf{B} = \mathbf{I}$, which means the state bits are used directly as random output bits, but in many generators most of the internal state bits are not sufficiently random. In these cases $r < n$, and either some state bits are not propagated through to the output, or multiple state bits are XOR'd together to produce each output bit.

The quality of a generators is measured in two ways: empirical tests, which look at generated sequences of numbers; and theoretical tests, which considered mathematical properties of the entire number sequence. Examples of empirical tests are to track the distribution of ones versus zeros, or to examine the Fourier transform of the uniform sequence, and to see if the observed behaviour deviates significantly from the behaviour of an ideal uniform source. Many such tests are gathered into batteries such as TestU01 [9], which examine long output sequences from a given RNG to find evidence of non-randomness.

Theoretical randomness of binary linear generators is measured using equidistribution [3], which describes how evenly distributed the output sequence is in multiple dimensions. Equidistribution is defined in terms of a resolution l, which is the number of most-significant bits considered from the output, and a dimension d, which is the number of consecutive output samples. If a generator is (d, l)-distributed then every possible length d sequence will be observed in the output sequence the same number of times. For example, if a 32-bit generator is $(3, 16)$-distributed then all possible 16-bit output triples will be equally likely. Alternatively, if a generator is (d, r)-distributed, i.e. it is equidistributed to d dimensions over the output bit-width, and it is initialised to an unknown random state x_0, then observing $y_1..y_{d-1}$ gives no information at all about what y_d will be. However, observing $y_1..y_d$ *may* allow us to make predictions about the value of y_{d+1}, or in some cases allow us to predict it precisely.

There are three stages when designing such a generator:

1) Describe a family of generators G_F, such that each member of G_F can be efficiently implemented in the target architecture. However, only *some* members of G_F will have the maximum period property.
2) Extract a maximum period subset $G_M \subset G_F$, such that all members of G_M implement a matrix $\mathbf{A}$ with a primitive characteristic polynomial. This is achieved either by randomly selecting and testing members of G_F, or by exhaustive enumeration if $|G_F|$ is small.
3) Find the generator $g_I \in G_M$ which produces the output stream with highest statistical quality, either by considering multiple members with different $\mathbf{A}$ matrices, or by trying many different $\mathbf{B}$ matrices for a single transition matrix.

The selected RNG instance $g_I \in G_F$ can then be expressed as code (e.g. C or VHDL) and used in the target architecture.

B. LUT-Optimised RNGs

LUT-optimised (LUT-OPT) generators [1] are a family of generators with a matrix A where each row and column contains $t - 1$ or t ones. In hardware terms this means that each row maps to a $t - 1$ or t-input XOR gate, and so can be implemented in a single t-input LUT. Thus if the current vector state is held in a register, each bit of the new vector state can be calculated in a single LUT, and an r-bit generator can be implemented in r fully-utilised LUT-FFs. The basic structure of a LUT-OPT generator is shown in Figure 1 (a).

A simple example of a maximum period LUT-OPT generator with $r = 6$ and $t = 3$ is given by the recurrence:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}, \quad \begin{bmatrix} x_{i+1,1} \\ x_{i+1,2} \\ x_{i+1,3} \\ x_{i+1,4} \\ x_{i+1,5} \\ x_{i+1,6} \end{bmatrix} = \begin{bmatrix} x_{i,2} \oplus x_{i,3} \\ x_{i,2} \oplus x_{i,3} \oplus x_{i,6} \\ x_{i,2} \oplus x_{i,4} \\ x_{i,1} \oplus x_{i,5} \\ x_{i,1} \oplus x_{i,6} \\ x_{i,1} \oplus x_{i,4} \oplus x_{i,5} \end{bmatrix}$$

Such matrices can be found for all $t \geq 3$ and $r \geq 4$, and are practical for generating up to ~ 1000 uniform bits per cycle. LUT-OPT generators have two key advantages:

- Resource efficiency: each additional bit requires one additional LUT and FF, so resource usage scales linearly - generating r bits per cycle requires r LUT-FFs.
- Performance: the critical path in terms of logic is a single LUT delay, so the generators are extremely fast: usually the clock net is the limiting factor, with routing delay and congestion only becoming a factor for large n.

However, these advantages are balanced by a number of disadvantages:

1) Complexity: Each (r, t) combination requires a unique matrix of connections, which must be found using specialised software. If these matrices are randomly constructed (as in previous work), then it is difficult to compactly encode these matrices, so it is difficult for FPGA engineers to make use of the RNGs.
2) Quality: The random bits are formed as a linear combination of random bits produced in the previous cycle - when $t = 3$ some of the new bits will be a simple 2-input XOR of bits from the previous cycle. The impact of this lag-1 linear dependence is minimal in modern FPGAs where $t \geq 5$, and also diminishes quickly as r is increased, but remains a source of concern.
3) Period: In order to achieve a period of $2^n - 1$ it is necessary to choose $r = n$, even if far fewer than n bits

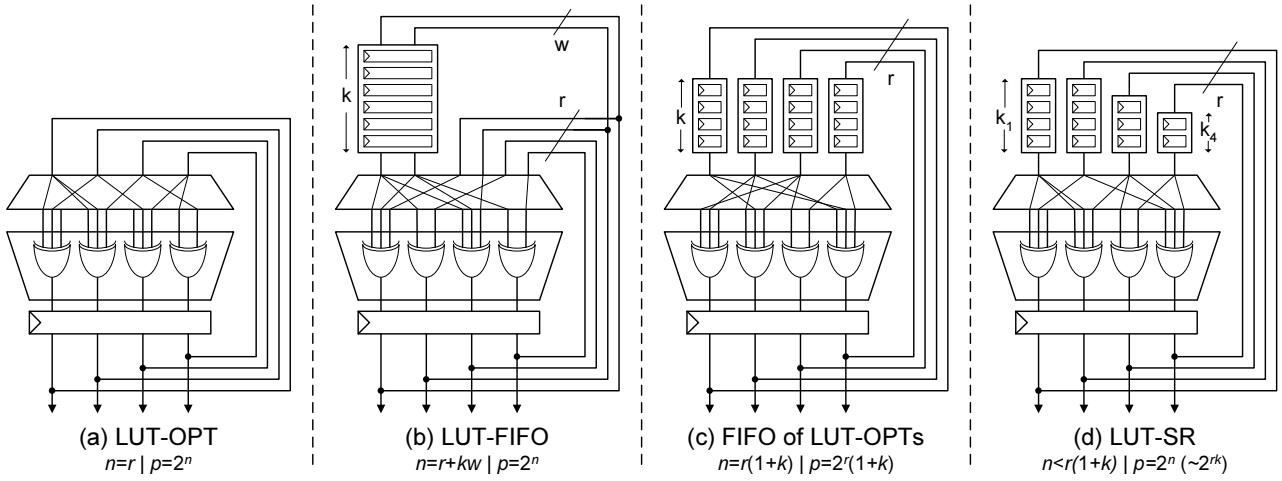

Fig. 1. Connection characteristics of four different types of FPGA optimised binary linear RNG.

are needed per cycle. An absolute minimum safe period for a hardware generator is $2^{64}-1$, but it is preferable to have much larger periods of $2^{1000}-1$ or more.

4) Seeding: It is necessary to initialise RNGs with a chosen state at run-time, so that different hardware instances of the same RNG algorithm will generate different random streams. In a LUT optimised generator it is *possible* to implement serial loading of state using one LUT input per RNG bit to select between RNG and load mode, but in practice for a randomly chosen matrix A only parallel loading is possible.

C. LUT-FIFO RNGs

One way of removing the quality and period problems is provided by LUT-FIFO generators [2]. These augment the r bits of state held in FFs with an additional depth-k width-w FIFO, for a total period of 2^n-1, where $n=r+wk$, shown in Figure 1 (b). LUT-FIFO generators can provide long periods such as $2^{11213}-1$ and $2^{19937}-1$, but also have disadvantages:

1) For reasonable efficiency the FIFO needs to be implemented using a block RAM, a relatively expensive resource which one would usually prefer to use elsewhere in a design.
2) The word-wise granularity of block-RAM based FIFOs reduces the flexibility in the choice of r, as it can only be varied in multiples of k.

These are mild disadvantages when compared to the quality and period problems of LUT optimised generators that have been eliminated, but LUT-FIFO generators also make the problems of complexity and efficient initialisation slightly worse. If extremely high quality and period are needed then LUT-FIFO generators present the fastest and most efficient solution, but few applications actually require such high levels of quality, particularly given the need for expensive block-RAM resources.

D. Software RNGs

In addition to the hardware-optimised LUT-OPT and LUT-FIFO generators, a number of generators designed for software architectures have been ported to FPGA architectures:

Combined Tausworthe [3] - Software generators which use word-level shift, XOR, and AND operations to construct simple recurrences with distinct periods, which are then combined using XOR to produce a much longer period generator.

Mersenne Twister [5] - This uses the same word-level operators as the Combined Tausworthe, combined with a large RAM-based queue, to create a software generator with fairly good equidistribution and the extremely long period of $2^{19937}-1$.

WELL [10] - This generator uses similar techiques to the Mersenne Twister, but uses a more complex recurrence step involving multiple memory accesses per sample, to achieve the maximum possible equidistribution at the same period as the Mersenne Twister.

All the software generators are designed with word-level instructions in mind, and so tend to be inefficient in terms of resources consumed per bit generated. In this paper we show the LUT-SR generator, which sits between the LUT optimised and LUT-FIFO generators: it fixes all problems related to complexity and serial seeding found with both generators, and provides much higher periods than LUT-OPT generators for a cost of 1 extra LUT-FF per bit, while eliminating the block-RAM resource needed for a LUT-FIFO RNG.

III. The LUT-SR RNG

Modern FPGAs allow LUTs to be configured in a number of different ways, such as basic ROMs, RAMs, and shift-registers. Configuring LUTs as shift-registers provides an attractive means of adding more storage bits to a binary linear generator: for example, adding one Xilinx SRL32 to a LUT-optimised r bit generator allows the state size to be increased to $n=r+32$. This represents a degenerate form of a LUT-FIFO generator, with $k=32$ and $w=1$.

However, while the FIFO in a LUT-FIFO RNG is usually an expensive block-RAM, LUT-based shift-registers are very cheap - almost as cheap as the LUTs used to build the XOR gates. So it now becomes economical to use r shift registers,

one per output bit, increasing the potential state to $n = r(1 + k)$. If we assume $k = 32$ (as found in modern FPGAs), and a modest RNG output width of $r = 32$, the state size increases to $n = 1056$. This provides a potential period of $2^{1056} - 1$ for a cost of 64 LUTs, as compared to a period of $2^{64} - 1$ for a LUT optimised generator with the same resource usage.

It might be tempting to simply configure all shift-registers with the same length, in an attempt to maximise the period for a given number of resources, but this cannot provide a maximum period generator. Instead it would result in $1 + k$ independent r-bit generators, with a sample taken from each on successive cycles, shown in Figure 1 (c). In LUT-FIFO generators this problem is avoided by making each new output bit dependent on one bit from the previous cycle, with the remaining $t - 1$ or $t - 2$ bits provided by the FIFO output. This lag-1 dependency is not ideal, but is generally benign as the LUT-FIFO uses deep block-RAM based FIFOs.

Bit-wide shift-registers enable a different solution which allows large periods to be achieved, while also improving the rate of mixing within the generator state. Each of the r shift-registers can be assigned some specific length $k_i \leq k$, reducing the state size to $n = \sum_{i=1}^{r}(1 + k_i)$. One solution would be to randomly configure each shift-register as $k_i = k$ or $k_i = k-1$, giving a period $rk < n < r(1+k)$. But a much more interesting solution is to randomly choose $1 < k_i \leq k$, subject to the constraint $\exists i, j : i \neq j \wedge \gcd(k_i+1, k_j+1) = 1$. This allows for much more rapid mixing between bits within the state, while still providing necessary (but not sufficient) conditions for mixing within the state.

Part (d) of Figure 1 shows the new LUT-SR style of generator. All four generators shown in the figure may seem superficially similar, but actually provide quite different trade-offs in terms of quality vs. resource usage.

Both the LUT-OPT and LUT-FIFO RNGs were originally developed and presented without much consideration of how to initialise the generator state. Initialisation is very important, as it is common to instantiate multiple parallel RNGs implementing the same algorithm, and each must be initialised with a distinct seed (initial state vector).

Given an l-input LUTs, we can implement a $t = l$ LUT-optimised RNG, but there is no way to read or write the RNG state. [2] Loading the state in parallel only allows a $t = l - 2$ RNG, as one LUT input is needed to select when to load, and the other is needed to provide the new 1-bit value. In a 4-LUT architecture, this implies $t = 2$, which is not possible (some bits would be direct copies of bits from the previous cycle), so two LUTs per bit would be needed. Parallel loading also means that the RNG initialisation circuitry (which is only needed at the start of simulations, and so should be as small as possible) must contain $O(r)$ resources, so even in the best-case the resources per RNG have doubled.

A better approach is to find a cycle through the matrix A, and use this to implement a 1-bit shift-register through the RNG. This only requires one bit per LUT (to select between RNG and load mode), so allows $t = l - 1$, requiring only one LUT per bit in a 4-LUT architecture, and allowing an efficient serial initialisation circuit, no matter how large r is. This approach can be extended to the LUT-FIFO RNG as well, as in any maximum period generator there must exist a cycle running through all state bits in the RNG.

[2]Excluding device-specific FF read/write chains, such as provided in Stratix.

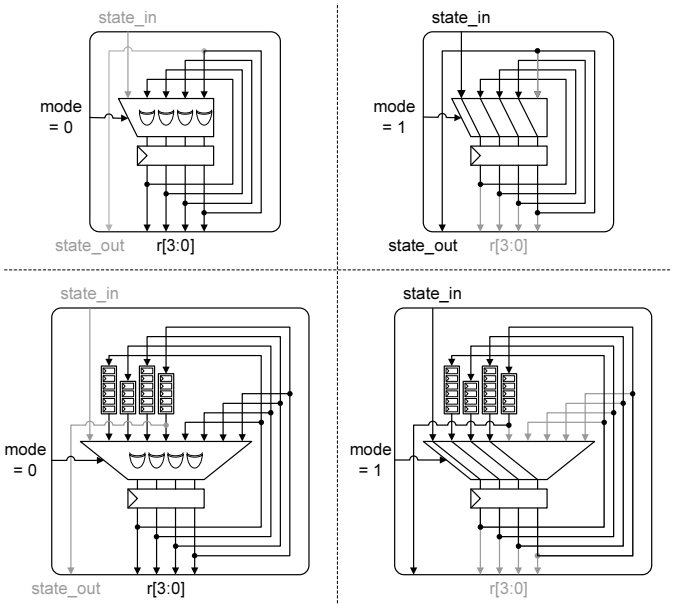

Fig. 2. Normal versus load paths for serial seeding of RNGs.

However, if the matrix A is randomly chosen, we must find a Hamiltonian cycle through a sparse graph of n points to determine the shift-register path. The original LUT-OPT paper correctly notes that this is *possible*, but even for reasonable values of n it can become computationally infeasible. In defining the LUT-SR generators, the provision of a serial load chain is explicitly taken into account, by embedding a chosen cycle into the matrix A from the start.

Specifically, we embed a very simply cycle of the form $i \leftarrow (i+1) \mod r$ through the XOR bits, shown in Figure 2. This simple cycle trivially extends through the shift-registers, and allows switching between the RNG and load modes using one input bit per LUT.

Including such a simple cycle in the generators could cause statistical problems for generators when $t = 3$, as there will be a simple linear dependence between adjacent output bits in cycles at a fixed lag. In an attempt to minimise this effect, an output permutation can be applied, to mix up the bits.

IV. Algorithm for Describing LUT-SR Generators

The broad class of LUT-SR generators as described is very general, and it is possible to construct a huge number of candidate LUT-SR RNGs by randomly generating binary matrices which meet the requirements. However, this presents the problem of communicating and disseminating RNGs experienced with LUT-OPT and LUT-FIFO papers: the matrices are far too large and complicated to be included in a publication, so must be provided separately from the paper. Instead the LUT-SR generator family uses a short but precise algorithm for expanding a tuple of five integers into the full RNG structure.

The algorithm takes as input a 5-tuple (n, r, t, k, s):

- n Number of state bits in the RNG (period is $2^n - 1$).
- r Number of random output bits generated per cycle.
- t XOR gate input count.
- k Maximum shift-register length.
- s Free parameter used to select a specific generator.

The first four parameters (n, r, t, k) describe the properties of the generator in terms of application requirements and architectural restrictions. The final parameter s is used to select from amongst one of 2^{32} candidates that the algorithm can produce with the chosen values of (n, r, t, k).

Note: arbitrary values of s will not *result in a valid RNG; the choice of s is critically dependent on (n, r, t, k), and modifying one or more components will break the generator. Please only use the tuples listed later in the paper.*

Listing 1 gives the expansion algorithm and RNG as a C++ class - the constructor takes a 5-tuple of RNG parameters and expands them into a complete description of the RNG, the "PrintConnections" function can then print an RTL style description of the RNG, while the "Step" function provides a software reference implementation.

The constructor expands the RNG using five stages:

1) *Create initial seed cycle*: A cycle of length r is created through the r XOR-gates at the output of the RNG. At this stage there are no FIFO bits, or equivalently there are r FIFOs of length 0.
2) *FIFO extension*: the cycle is randomly extended until a total cycle length of n is reached, by randomly selecting a FIFO and increasing its length by 1, while maintaining the known cycle.
3) *Add loading connections*: the known cycle is added to the graph "taps", which describes the matrix A. The cycle describes the FIFO connections completely, and also describes the first input to each of the r XOR gates.
4) *Add XOR connections*: the cycle provides one input for each of the XOR gates, so now the additional $t - 1$ random inputs are added over $t - 1$ rounds. Each round is constructed from a permutation of the FIFO outputs, which ensures that at the end each FIFO output is used at most t times. Some bits will be assigned the same FIFO bit in multiple rounds, and so will have fewer than t inputs: this is critical to achieve a maximum period generator, and also provides us with an entry point into the cycle for seed loading.
5) *Output permutation*: the simple dependency between adjacent bits is masked using a final output permutation.

After construction, the RNG structure can be printed as RTL-style C using the function "PrintConnections". For example, the input tuple $(n = 12, r = 4, t = 3, k = 3, s = 0x4d)$ describes a generator with period $2^{12} - 1$, producing 4 random bits per cycle, with a maximum of 3 inputs per XOR gate, and a maximum FIFO depth of 3. The re-formatted and commented output is shown in Figure 3;

Due to the very simple printing code the format is not very pretty, but this describes a generator in terms of six variables:

ns,cs : cs is the current state of the generator, and ns is the next state of the generator. Both are n bit vectors, and describe both the FF and FIFO state.

Listing 1. Source code for decoding generators

```
struct rng{
 static int LCG(uint32_t &s) // Simple LCG RNG
 { return (s=1664525UL*s+1013904223UL)>>16; }

 static void Permute(uint32_t &s, vector<int> &p)
 { for(int j=p.size();j>1;j--) swap(p[j-1],p[LCG(s)%j]); }

 int n, r, t, maxk;   // rng parameters
 uint32_t s;  // Seed for generator
 vector<set<int> > taps; // connections
 vector<int> cycle; // cycle through bits
 vector<int> perm; // output permutation
 int seedTap;   // Entry point to cycle

 rng(int _n, int _r, int _t, int _maxk, uint32_t _s)
  : n(_n), r(_r), t(_t), maxk(_maxk), s(_s)
  , taps(n), cycle(n), perm(r), seedTap(0)
 {  // Construct an rng using (n,r,t,maxk,s) tuple
  vector<int> outputs(r), len(r,0);    int bit;

  // 1: Create cycle through bits for seed loading
  for(int i=0;i<r;i++){ cycle[i]=perm[i]=(i+1)%r; }
  outputs=perm; // current output of each fifo

  for(int i=r;i<n;i++){ // 2: Extend bit-wide FIFOs
   do{ bit=LCG(_s)%r; }while(len[bit]>=maxk) ;
   cycle[i]=i;       swap(cycle[i], cycle[bit]);
   outputs[bit]=i;    len[bit]++;
  }

  for(int i=0;i<n;i++) // 3: Loading connections
   taps[i].insert(cycle[i]);

  for(int j=1;j<t;j++){ // 4: XOR connections
   Permute(_s, outputs);
   for(int i=0;i<r;i++){
    taps[i].insert(outputs[i]);
    if(taps[i].size()<taps[seedTap].size())
     seedTap=i;
  }}

  Permute(_s, perm); // 5: Output permutation
 }

 void PrintConnections() const
 {  // Dump transition function in "C" format
  for(int i=0;i<n;i++){
   // Create connections for load mode
   if(i==seedTap) printf("ns[%u]=m?s_in:(0", i);
   else printf("ns[%u]=m?cs[%u]:(0",i,cycle[i]);

   // Create XOR tree for RNG mode
   set<int>::iterator it=taps[i].begin();
   while(it!=taps[i].end()) printf("^cs[%u]",*it++);
   printf(");\n");
  }
  printf("s_out=cs[%u];\n", cycle[seedTap]);

  for(int i=0;i<r;i++) // output permutation
   printf("ro[%u]=ns[%u];\n", i, perm[i]);
 }

 pair<vector<int>,int> // returns (ro[0:r-1],s_out)
  Step(vector<int> &cs, int m, int s_in) const
 { // Advance state cs[0:n-1] using inputs (m,s_in)
  vector<int> ns(n, 0), ro(r);

  for(int i=0;i<n;i++){ // Do XOR tree and FIFOs
   if(m==0){ // RNG mode
    std::set<int>::iterator it=taps[i].begin();
    while(it!=taps[i].end()) ns[i] ^= cs[*it++];
   }else{ // load mode
    ns[i]= (i==seedTap) ? s_in : cs[cycle[i]];
  }  }

  // capture permuted output signals
  int s_out=cs[cycle[seedTap]]; // output of load chain

  cs=ns; // "clock-edge", so FFs toggle
  for(int i=0;i<r;i++) ro[i]=cs[perm[i]];
  return make_pair(ro,s_out);
 }
};
```

```
// Sequential update of registers using XOR gates
ns[0]= m?s_in  :(0^cs[9]^cs[10]); // Load chain input
ns[1]= m?cs[6] :(0^cs[6]^cs[11]);
ns[2]= m?cs[11]:(0^cs[6]^cs[10]^cs[11]);
ns[3]= m?cs[9] :(0^cs[9]^cs[10]^cs[11]);
// Sequential update of internal Shift-Register bits
ns[4]= m?cs[3] :(0^cs[3]);
ns[5]= m?cs[1] :(0^cs[1]);
ns[6]= m?cs[2] :(0^cs[2]);
ns[7]= m?cs[0] :(0^cs[0]);
ns[8]= m?cs[5] :(0^cs[5]);
ns[9]= m?cs[7] :(0^cs[7]);
ns[10]=m?cs[8] :(0^cs[8]);
ns[11]=m?cs[4] :(0^cs[4]);
// Combinatorial outputs
s_out =cs[10];  // Output of serial load chain
ro[0] =ns[3]; // Permuted output bits.
ro[1] =ns[2];
ro[2] =ns[0];
ro[3] =ns[1];
```

Fig. 3. Pseudo-RTL output from "PrintConnections" function for tuple $(n = 12, r = 4, t = 3, k = 3, s = 0x4d)$.

m,s_in : These are the RNG inputs, with m choosing between RNG mode (m=0), and load mode (m=1); s_in provides the serial load input in load mode.

ro : This is the r-bit random output of the generator, which is simply a permutation of the first r bits of the generator state.

s_out : While loading a new state using m=1, this signal can be used to read the current state.

The function "Step" implements the same transition function directly in C++, and can operate in both RNG and load mode.

The aim of this algorithm is to provide a precise specification that can be included in the paper. A more complete package of tools is available at http://www.doc.ic.ac.uk/~dt10/research/rngs-fpga-lut_sr.html.This includes functions for generating platform-independent VHDL code, which also extract the logical shifters to improve compilation performance. [3] In addition, the package provides test-bench generation tools, which verify the VHDL code against the reference software, and demonstrate how to use the serial load capability.

V. LUT-SR Search Process

Algorithm 1 can expand any given tuple into a candidate generator, but only specific parametrisations produce maximum length generators, and even amongst maximum length generators some have better quality than others.

A given 4-tuple (n, r, t, k) defines a set of 2^{32} candidate 5-tuples (as s is 32-bit) which can be explored by varying s exhaustively or randomly, but there is no guarantee that a valid s can be found for any given 4-tuple - we might look at all 2^{32} tuples and find none which are maximum period. For a given n, there exist $\varphi(2^n - 1)/n$ distinct degree-n primitive polynomials, where $\varphi(.)$ is Euler's totient function. Given there are 2^n possible polynomials (including those of degree less than n), this means the probability of a random polynomial being primitive is:

$$\varphi(2^n - 1)2^{-n-1} \tag{3}$$

While the algorithm may generate 2^{32} candidates, there is no guarantee that the candidates all have distinct random polynomials. Assuming each candidate samples represents a random sample from the set of 2^n possible polynomials, the expected number of distinct candidate polynomials is:

$$2^n \left(1 - \left[\frac{2^n - 1}{2^n}\right]^{2^{32}}\right) > 0.6 \min(2^n, 2^{32}) \tag{4}$$

The expected primitive generators per 4-tuple is then simply the probability of any polynomial being primitive times the expected number of distinct polynomials in the 4-tuple's candidate set.

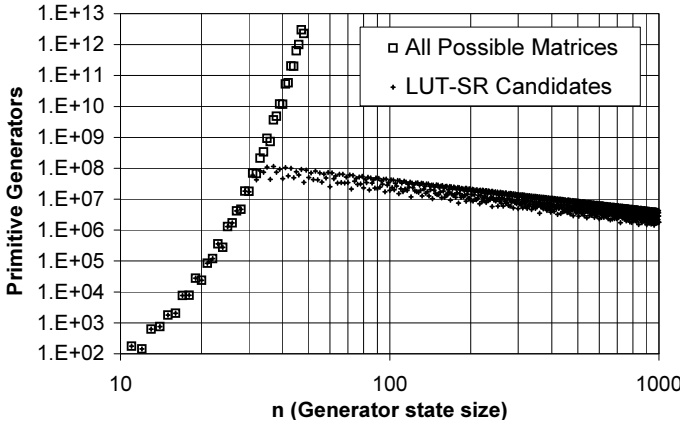

Fig. 4. Expected number of primitive generators for increase generator state size.

Figure 4 shows the number of expected primitive generators for a given n, within both the LUT-SR candidate set and the set of all possible $n \times n$ transition matrices. For $n < 32$ the number of generators is limited by the small number of possible polynomials, and many values of s will lead to the same polynomial. As n exceeds 32, the number of possible primitive polynomials increases greatly, but the number in the candidate set slowly decreases. [4] However, the rate of decline is slow enough that there are plenty of candidates in the set: when $n = 10^3$ there are over a million primitive candidates, and when $n = 10^4$ there are still hundreds of thousands.

Having established there are plenty of primitive generators within the candidate set, the problem is now how to find them, i.e. to implement Stage 2 of the process described in Section II-A. Practically, this means we must check whether a given concrete 5-tuple has a primitive characteristic polynomial. One approach is to build the set of feed-back taps using the LUT-SR algorithm and use them to create the matrix $\mathbf{A}$, from which we can then extract the characteristic polynomial. However, calculating the characteristic polynomial of an arbitrary matrix is an $O(n^3)$ operation, so for large n this becomes impractical given the need to check thousands of random matrices.

[3] Synthesis tools can extract the shifters from the output of "PrintConnections", but take a long time for large n.

[4] The trend continues in the same way beyond $n = 1000$, but is not shown to simplify the graph.

Instead of using the matrix form, it is more efficient to extract the characteristic polynomial from the output sequence. After building the candidate generator, the generator is put into an arbitrary non-zero state and the "Step" function is used to generate a sequence of $2n$ consecutive outputs. The Berlekamp-Massey algorithm is then used to extract the minimal polynomial from this sequence - while there are some differences between the minimal and characteristic polynomial, for the purposes of testing whether a given generator is maximum period they can be used in the same way.

For the generator to be full-period the characteristic $P(z)$ must be a degree-n primitive polynomial. Primitive polynomials must be irreducible, that is they should not factor into smaller polynomials (a similar concept to primality for integers). If 2^n-1 is a Mersenne prime, then irreducibility is sufficient to prove primitivity. However, if 2^n-1 is composite, then it is also necessary to show for all prime factors f of 2^n-1 that $P(z)^{(2^n-1)/f} \neq P(z)$. This places limits on which specific values of n can be chosen: if the factorisation of 2^n-1 is not currently known, then it cannot be used to create a maximum period generator.

The process for testing whether a given 5-tuple is a primitive generator then becomes:

1) Expand the 5-tuple into the set of feedback taps.
2) Choose a random starting state, and generate a sequence of $2n$ successive outputs from the generator.
3) Use the Berlekamp-Massey algorithm to extract the minimal polynomial $P(z)$ from the sequence.
4) If $P(z)$ has degree less than n then **reject**.
5) If $P(z)$ is reducible, then **reject**.
6) For each prime factor f of 2^n-1, if $P(z)^{(2^n-1)/f} = P(z)$ then **reject**.
7) The polynomial is primitive, so **accept**.

Any 5-tuple which completes this process has a known maximum period of 2^n-1, so now we need to select the best from amongst this set.

VI. Quality Selection Process

The quality criterion we use to select the best value of s is based on equidistribution (see Section II-A). If a generator is (d,l)-distributed, then it is also $(d-1,l)$-distributed and $(d,l-1)$-distributed, where $d \geq 1$ and $l \geq 1$. The maximum dimension to which a generator is distributed to l-bits is d_l, with an upper bound of $d_l^* = \lfloor n/l \rfloor$. This leads to two common quality measures [3] based on the dimension gap $\delta_l = d_l^* - d_l$:

$$\Delta_1 = \sum_{i=1}^{r} \delta_l, \qquad \Delta_\infty = \max_{i=1..r} \delta_l \tag{5}$$

When a generator has $\Delta_1 = \Delta_\infty = 0$ it is *maximally equidistributed*, as it has achieved the best possible equidistribution for any generator with the same n and r.

However, these metrics suffer from problems due to the reliance on absolute dimension differences. For example, if $n = 64$ then the dimension gap $\delta_2 = 1$ means that the generator achieved a dimension of 31 rather than 32, but the same size gap at $\delta_{32} = 1$ means that the generator has a dimension of 1 rather than 2. The first gap is not that significant, but the second gap means the quality has effectively halved, making it difficult to use dimension gaps to compare generators with the same n. Similarly, one cannot easily compare generators with different n: a value of $\Delta_\infty = 1$ is very good for a generator with $n = 19937$, but is very bad for a generator with $n = 32$.

The metric we used when selecting from amongst the maximum period generators found is a metric called Q:

$$Q_l = \sqrt[l]{\prod_{i=1}^{i=l} \frac{d_l}{d_l^*}} \tag{6}$$

This metric produces a value in the range $(0,1]$, with $Q_r = 1$ occurring only for maximally equidistributed generators. It also takes into account relative changes in dimension: for $\{n = 64, r = 32, \Delta_1 = 1\}$, if $\delta_2 = 1$ then $Q_{32} = 0.999$, while if $\delta_{32} = 1$ then $Q_{32} = 0.978$, making it clear which one is better.

There are two main ways of calculating equidistribution, either by using lattice reductions in terms of the output sequence [7], or by using matrix operations on the state sequence [3]. The lattice methods are often faster for evaluating software generators, as they work well when $r \ll n$, and the software generators are able to generate output samples efficiently. However, generating samples from a generator optimised for hardware is relatively slow due to the complicated bit-wise recurrence, while the larger r values reduce efficiency.

The matrix based methods operate by creating a matrix which relates the dl bits of the d-tuple to the current state of the generator. The tuple we are measuring is:

$$\mathbf{e}_{i,d,l} = (\mathbf{y}_{i,1..l}, \mathbf{y}_{i+1,1..l}, ..., \mathbf{y}_{i+d,1..l}) \tag{7}$$

$$= (\mathbf{S}_l\mathbf{B}\mathbf{x}_i, \mathbf{S}_l\mathbf{B}\mathbf{A}\mathbf{x}_i, \mathbf{S}_l\mathbf{B}\mathbf{A}^2\mathbf{x}_i, ..., \mathbf{S}_l\mathbf{B}\mathbf{A}^{d-1}\mathbf{x}_i) \tag{8}$$

where S_l is an $l \times n$ matrix containing the first l rows of the identity matrix. A $dl \times n$ matrix $\mathbf{E}$ can be constructed which maps the bits of x_i to the dl bits of e_i:

$$\mathbf{E}_{d,l} = \begin{bmatrix} (\mathbf{S}_l\mathbf{B})^T & (\mathbf{S}_l\mathbf{B}\mathbf{A})^T & ... & (\mathbf{S}_l\mathbf{B}\mathbf{A}^{d-1})^T \end{bmatrix}^T \tag{9}$$

If and only if the matrix $E_{d,l}$ has full-rank then the generator is (d,l)-distributed.

In principle the matrix-based method requires $O(n^4)$ time, due to the need to perform $O(n)$ matrix multiplications, but in practise $\mathbf{A}$ is extremely sparse for LUT-SR generators, so the matrix-multiply cost is $O(n^2)$ and the actual cost of forming $E_{d,l}$ is effectively $O(n^3)$. The matrix rank calculation can also be performed progressively, so the echelon form of $E_{d,l}$ can be used as the starting point for echelonising $E_{d+1,l}$, allowing d_l to be iteratively determined starting from dimension 1.

The generator search and equidistribution calculations were performed using a combination of NTL [11] and M4RI [12] for dense matrix and polynomial operations; a customised version of PPSearch [13] for primitivity testing; a custom sparse-matrix library optimised for random number generator operations; and a custom progressive rank library for equidistribution calculations.

n	r	$t=3$	$t=4$	$t=5$	$t=6$
1024	32	1a5eb	1562cd6	1c48	2999b26
1280	40	c51b5	4ffa6a	3453f	171013
1536	48	76010	c2dc4a	4b2be0	811a15
1788	56	a2aae	23f5fd	1dde4b	129b8
2048	64	5f81cb	456881	bfbaac	21955e
2556	80	755bac	7454a5	8a0c78	cc7516
3060	96	79e56	9a7cd	41a62	1603e
3540	112	78d9df	7737bf	870295	b850c9
3900	128	10023	197bf8	cc71	14959e
5064	160	42f017	3d31e4	43c621	51249a
5064	192	48a92	439d3	4637	577ce
6120	224	3e2834	3ca4af	401dfd	42d8f2
8033	256	437c26	439995	43664f	427ba2
11213	384	a6847	92228	a4afa	afd67
19937	624	209eb	2e5fa	2fffb	25c7d

TABLE I
TABLES OF s FOR RNGS OF FORM $(n, r, t, k = 32, s)$.

VII. TABLES OF LUT-SR GENERATORS

Table I provides a list of generator tuples for a variety of useful parametrisations. To provide maximum real-world benefit, we have chosen to examine the situation where $k = 32$. This works well in modern FPGAs, requiring one LUT per shift-register, and means that each generator needs $2r$ LUTs and r FFs. Using the shift-register output directly frees up the associated FF, but reduces clock rate slightly. For maximum speed the final output of the shift-register can be placed in a FF, increasing the resource usage to $2r$ LUT-FFs, while allowing 600MHz+ performance in Virtex-6 without any manual tuning.

We choose $n \sim rk$, as this means that the period increases with the number of output bits, and results in a similar equidistribution, even when large numbers of bits are generated per cycle. Note that because we need to know the factorisation of $2^n - 1$ (see Section V) we cannot choose $n = rk$ when rk becomes large, so instead choose the closest pair for which the factorisation is known.

Input taps for $t = 3..6$ are considered, as it may be preferable to use more or less taps depending on the situation. For example, a Virtex-4 generator could be implemented using $t = 3$ and two SRL16s per shift-register, or an RNG that doesn't need to be explicitly seeded could be implemented using $t = 6$ in a Virtex-6. In general $t = 5$ is recommended, as this provides a high-level of state mixing, while still allowing an efficient implementation in modern architectures.

The generators listed in the table have been tested using the Crush and BigCrush empirical test batteries from the TestU01 package [9]. These batteries perform extremely stringent statistical tests, and the LUT-SR generators pass all of them convincingly, *except* the matrix rank and linear complexity tests. This is a known problem with all binary linear generators, including the Mersenne Twister, WELL, LFSRs, and Combined Tausworthe generators, all of which will fail such tests. However, in practise these specific flaws rarely affect simulations: the Mersenne Twister has been used in thousands of applications without problems, so the same should be true of the LUT-SR generators.

Generator	Failed Tests	n	r	$\frac{w(P_z)}{n}$	RAM	LUT	FF	$\frac{r}{\text{LUT}}$
TAUS-113 [3]	6	113	32	0.43	0	87	208	0.37
TT-800 [14]	14	800	32	0.33	2	162	162	0.26
MT-19937 [15]	2	19937	32	0.01	2	278	-	0.12
WELL-19937 [16]	2	19937	32	0.43	4	633	537	0.05
LFSR-160 [1]	13	160	32	0.03	0	448	384	0.07
LUT-OPT [1]	4	1024	1024	0.23	0	1024	1024	1.00
LUT-FIFO [2]	2	11213	521	0.50	1	539	611	0.97
LUT-SR (t=5)	4	1024	32	0.45	0	64	64	0.50
LUT-SR (t=5)	2	19937	624	0.50	0	1248	1248	0.50

TABLE II
COMPARISON OF GENERATORS BY QUALITY AND RESOURCE USAGE.

VIII. EVALUATION OF LUT-SR GENERATORS

Table II provides a comparison of a number of random number generators suggested for FPGAs. The TAUS-113 [3], TT-800 [14], MT-19937 (Mersenne Twister) [15], and WELL-19937 [16] generators are all software generators ported to hardware, while the LUT-OPT [1], LUT-FIFO [2], and LUT-SR are all designed specifically for FPGAs. The LFSR-160 uses 32 bit-wide LFSRs in parallel, which has an efficient implementation in both hardware and software.

The table includes three quality metrics: "Failed Tests" is the number of tests from the BigCrush battery that are failed; "n" gives the period of the generators; and $w(P_z)/n$ is the ratio of ones to zeros in the characteristic polynomial. All else being equal, larger n implies higher quality, while $w(P_z)/n$ should be relatively close to 0.5, but these are only relative metrics which are difficult to interpret on an absolute scale.

The number of tests failed in BigCrush is a more useful absolute metric of quality, as the tests detect specific quality problems in the generator. TT-800 and LFSR-160 do particularly badly, failing multiple tests, including those which do *not* depend on the linear structure of the generator – that is, non-randomness beyond that required by the generation method. All the other methods only fail the Linear-Complexity and Matrix-Rank tests, with the difference in numbers of failures due to the differences in periods: BigCrush uses a number of different parametrisations of the tests, and longer period generators are able to pass tests that look for predictable linear complexity in smaller sub-sequences.

Figure 5 compares the equidistribution of the different generators over the first 32 bits. Looking at the long period generators, WELL-19937 and LUT-SR-19937 have effectively the same equidistribution, although LUT-SR-19937 does not quite achieve the Maximally Equidistributed property – it achieves $\Delta_\infty = 1$ and $\Delta_1 = 3$ over the first 32 MSBs; considering greater resolutions, $\delta_4 = \delta_8 = \delta_{15} = \delta_{66} = \delta_{89} = 1$, and for the first 128 bits we have $\Delta_\infty = 1$ and $\Delta_1 = 5$. This equidistribution is considerable better than MT-19937, which develops large enough dimension gaps at 16 bits to drop to the quality of the much lower period LUT-FIFO-11213.

In the lower-period group, the LUT-OPT-1024 generator doesn't quite achieve the ME property but still has $\Delta_\infty = 1$ over the first 32 bits, while the LUT-SR-1024 generator does not do as well. For the first 10 bits it is ME, but then develops a small dimension gap at 10 bits followed by a steeper drop at

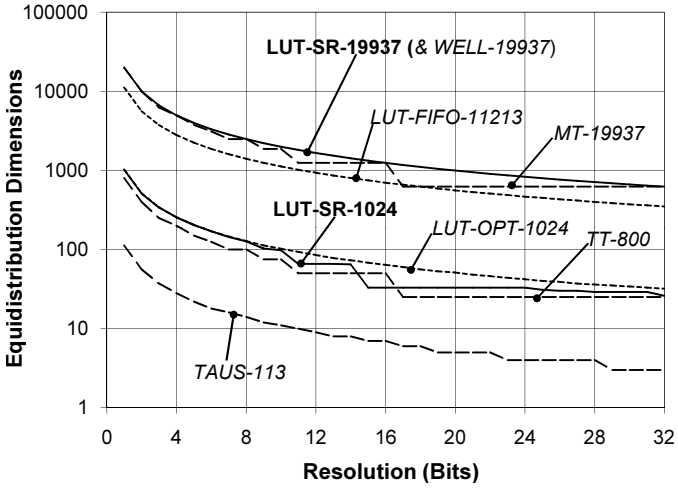

Fig. 5. Number of equidistributed dimensions (quality) for increasing resolution (number of MSBs).

15 bits. However, it is distributed to 22 dimensions at 32-bit resolution, an order of magnitude more than TAUS-113.

The FPGA-optimised generators all provide the best performance in terms of quality vs resources: amongst the lower period generators the LUT-OPT generator uses the absolute minimum resources of one LUT per generated bit, but unless one wishes to use all 1024 generated bits it is far less efficient than the LUT-SR-1024; the LUT-FIFO generator can provide very long periods to match those of the Mersenne Twister, but requires the use of a block RAM; and now the new LUT-SR generator provides a useful mid-point between the two, with a good balance between resource utilisation and good quality.

The software RNGs provide the best equidistribution, but it is difficult to justify the small improvement in quality over the LUT-SR RNGs given the cost in resources; the WELL-19937 generator requires four block-RAMs to remove a dimension gap of $\Delta_1 = 3$, and only achieves 1/19th of the throughput. The ability to customise the number of generated bits is also a particular advantage in FPGA applications, as it is rare that precisely 32-bits are needed.

In terms of performance, all the FPGA-optimised generators are also intrinsically fast: both the LUT-OPT and LUT-SR generators have a LUT-FF-LUT critical path, and provide post-place-and-route clock rates of 600MHz+ in Virtex-6 without any optimisation, even for large values of r. The LUT-FIFO generators are typically limited by the clock rate of the block-RAM providing the FIFOs, and so can achieve 550MHz+.

IX. Conclusion

This paper presents a family of FPGA-optimised uniform random number generators, called LUT-SR RNGs. These RNGs takes advantage of the ability to configure LUTs as independent shift-registers, allowing high-quality long period generators to be implemented using only a small amount of logic. In addition the period and quality scale with the number of output bits, unlike generators adapted from software.

A key advantage of the LUT-SR generators over previous FPGA-optimised uniform random number generators is that they can be reconstructed using a simple algorithm, contained in the paper. In concert with the tables of maximum period generators, this allows FPGA engineers to use the new RNGs without needing to find generator instances themselves.

References

[1] D. B. Thomas and W. Luk, "High quality uniform random number generation using LUT optimised state-transition matrices," *Journal of VLSI Signal Processing*, vol. 47, no. 1, pp. 77–92, 2007.

[2] ——, "FPGA-optimised high-quality uniform random number generators," in *Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2008, pp. 235–244.

[3] P. L'Ecuyer, "Tables of maximally equidistributed combined LFSR generators," *Mathematics of Computation*, vol. 68, no. 225, pp. 261–269, 1999.

[4] D. B. Thomas and W. Luk, "FPGA-optimised uniform random number generators using luts and shift registers," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2010, pp. 77–82.

[5] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[6] M. Saito and M. Matsumoto, "SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator," in *Monte-Carlo and Quasi-Monte Carlo Methods*, 2006, pp. 607–622.

[7] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Transactions on Mathematical Software*, vol. 32, no. 1, pp. 1–16, 2006.

[8] M. Matsumoto and Y. Kurita, "Twisted GFSR generators II," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 3, pp. 254–266, 1994.

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

[10] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Transactions on Mathematical Software*, vol. 32, no. 1, pp. 1–16, 2006.

[11] V. Shoup, "NTL: A library for doing number theory," http://www.shoup.net/ntl/.

[12] M. Albrecht and G. Bard, *The M4RI Library – Version 20100817*, The M4RI Team, 2010. [Online]. Available: http://m4ri.sagemath.org

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

[14] 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*, 2007, pp. 529–532.

[15] S. Konuma and S. Ichikawa, "Design and evaluation of hardware pseudo-random number generator mt19937," *IEICE - Trans. Inf. Syst.*, vol. E88-D, no. 12, pp. 2876–2879, 2005.

[16] Y. Li, P. C. J. Jiang, and M. Zhang, "Software/hardware framework for generating parallel long-period random numbers using the well method," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2011, pp. 110 –115.

David Thomas (M06) received the M.Eng. and Ph.D. degrees in computer science from Imperial College London, in 2001 and 2006, respectively, Since 2010, he has been a Lecturer with the Electrical and Electronic Engineering Department, Imperial College London. His research interests include hardware-accelerated cluster computing, FPGA-based Monte Carlo simulation, algorithms and architectures for random number generation, and financial computing.

Wayne Luk Wayne Luk (F09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford. He is a Professor of computer engineering with Imperial College London, and a Visiting Professor with Stanford University and Queens University Belfast. His research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.