FPGA-Optimised Uniform Random Number Generators using LUTs and Shift Registers

David B. Thomas and Wayne Luk
Imperial College London
{dt10,wl}@doc.ic.ac.uk

***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 new 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-quality LUT-FIFO RNGs and low-resource low-quality LUT-OPT RNGs. 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.**

I. Introduction

Monte Carlo applications are ideally suited to FPGAs, due to the embarrassingly 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 (hopefully) makes it just as easy to use the FPGA-optimised generators, by providing a simple method for engineers to create a generator for many parametrisations. Specifically, it shows how to create a new type of generator with high quality and long periods, by using LUTs as shift-registers. The main contributions are:

- A new 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} and more using two LUTs and two FFs per generated random bit.
- An algorithm for describing LUT-SR RNGs using five integers.
- A minimal but complete C++ executable specification for expanding and simulating the generators included in the paper, backed by an on-line set of test-benches and tools.
- Tables of LUT-SR RNGs for output widths from 32 up to 624, with periods from 2^{1024} up to 2^{19937}.

This paper concentrates on presenting the high-level design properties and practical features of the new LUT-SR generators, with the aim of making them immediately available for use by academic and industrial users. Some mathematical properties are simply stated without a detailed explanation of how they were calculated, as most readers will only be interested in the practical results; the detailed theoretical background will be fully presented in a future journal paper.

II. Overview of Binary Linear RNGs

The new RNGs introduced here 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 (MT19937) [4], the Combined Tausworthe (Taus113) [3], and TT800 [5]. Many of these software generators have been adapted for use in FPGAs [6]–[8], but the convenience of mapping an algorithm designed for word-level software comes at the cost of reduced efficiency and flexibility.

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 ($\otimes$). The recurrence of an RNG is defined as:

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

where $\mathbf{x}_i = (x_{i,0}, x_{i,1}, ..., x_{i,n-1})^\mathsf{T}$ is the n-bit state of the generator, and $\mathbf{A}$ is an $n \times n$ binary transition matrix. Because the state is finite, and the recurrence is deterministic, eventually the 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].

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 sub-set $G_M \subseteq G_F$, such that all members of G_M implement a matrix 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 enough.
3) Find the generator $g_I \in G_M$ which produces the output stream with highest statistical quality; usually a theoretical metric called "equidistribution" is used to determine generator quality [3].

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, 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:

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

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 is a single LUT delay, so the generators are extremely fast (the clock net is usually the limiting factor).

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 it is necessary to choose $r = n$, even if far fewer than n bits are needed per cycle. An absolute minimum safe period for a hardware generator is 2^{64}, but it would be preferable to have much larger periods of 2^{1000} 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-k FIFO, for a total period of $n = r + wk$, shown in Figure 1 (b). LUT-FIFO generators can provide very long periods such as 2^{11213} and 2^{19937}, but have their own 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 for most applications such a generator is complete overkill, and a waste of block-RAM resources.

In this paper we develop an RNG 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 quality and periods than LUT-OPT generators for a cost of 2 LUT-FFs per bit, while eliminating the block-RAM resource needed for a LUT-FIFO RNG.

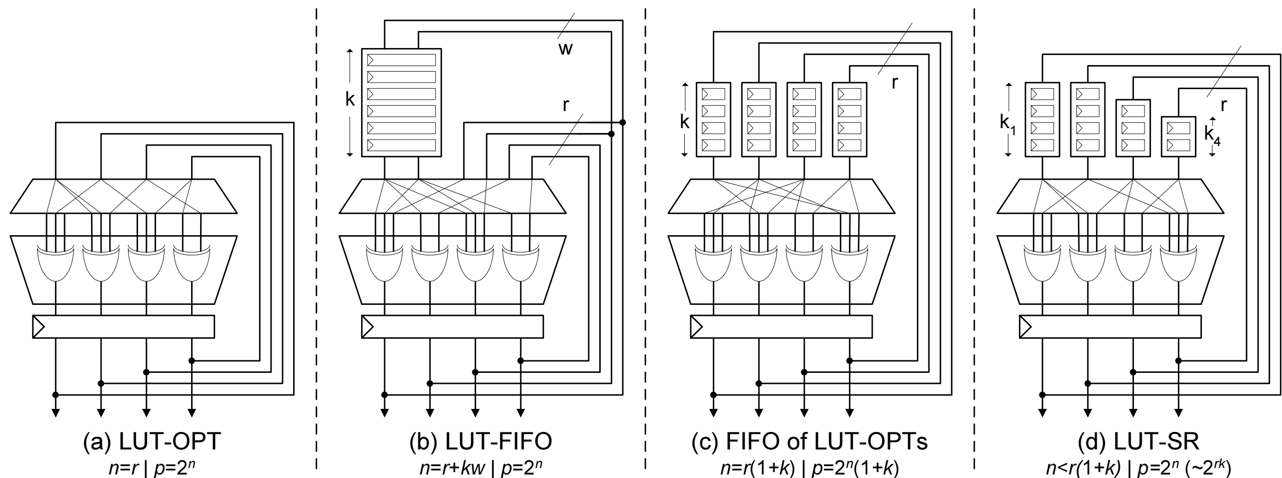

Fig. 1. Connection characteristics of four different types of FPGA optimised binary linear RNG.

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} for a cost of 64 LUTs, as compared to a period of 2^{64} 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. [1] 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 there must exist a cycle running through all state bits in the RNG.

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

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.

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. This paper uses a different approach, and provides 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.

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[perm[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);
 }
};
```

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 resulting output is:

```
ns[0]=m?s_in:(0^cs[9]^cs[10]);
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]);
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]);
s_out=cs[10];
ro[0]=ns[3];
ro[1]=ns[2];
ro[2]=ns[0];
ro[3]=ns[1];
```

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.
- 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/d̃t10/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. [2] 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.

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

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 $(n, r, t, k = 32, s)$.

V. TABLES OF LUT-SR GENERATORS

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. 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 800MHz+ 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 (a form of theoretical quality measure), even when large numbers of bits are generated per cycle. Note that for mathematical reasons it is difficult to choose $n = rk$ when rk becomes large [2], so for large numbers of output bits some "strange" values of r and n are chosen.

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.

VI. EVALUATION OF LUT-SR GENERATORS

The generators listed in the table have been tested using the Crush and BigCrush empirical test batteries from

Generator	Quality	n	r	$w(P_z)$	RAM	LUT	FF	r/LUT
Taus113 [3]	Medium	113	32	49	0	87	208	0.37
TT800 [7]	Medium	800	32	261	2	162	162	0.26
MT19937 [10]	Good	19937	32	135	2	278	-	0.12
LFSR-160 [1]	Poor	160	32	5	0	448	384	0.07
LUT-OPT [1]	Medium	512	512	235	0	513	512	1.00
LUT-FIFO [2]	Good	11213	89	5299	1	115	181	0.77
LUT-SR (t=5)	Good	1024	32	461	0	64	64	0.50

TABLE II
COMPARISON OF GENERATORS BY QUALITY AND RESOURCE USAGE.

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, 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.

The quality of software generators is usually measured using equidistribution [3], but the software notion of word-based equidistribution is less useful when considering hardware applications, which can easily re-group random bits into integers of different sizes. The most immediately useful measure of quality for LUT-SR generators is to determine the number of dimensions d to which a generator is r-distributed. The simplest interpretation is that if a generator is r-distributed to d dimensions, then it is impossible to predict anything about the next RNG output from all bits generated in the previous $d-1$ RNG outputs; alternatively, each d-tuple of r-bit integers is produced with equal likelihood.

All LUT-SR generators listed here are r-distributed to *at least* 20 dimensions, and most achieve 24 or more. This level of quality is automatically maintained as r increases, because the period scales with the number of output bits. By comparison, the Combined Tausworthe generator Taus-113 (recently popular for use in FPGAs), is only r-distributed over 3 dimensions, while an array of 32 parallel 160-bit LFSRs *might* be r-distributed over 5 dimensions if they are carefully initialised (if randomly initialised it will probably be lower).

Table II provides a comparison of a number of random number generators suggested for FPGAs. The Taus113 [3], TT800 [7], and MT19937 (Mersenne Twister) [10] generators are all software generators implemented in 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 FPGA-optimised generators all provide the best performance in terms of quality vs resources: amongst the low quality generators the LUT-OPT generator uses the absolute minimum resources of one LUT per generated bit, but cannot provide long periods or high quality; 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. 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 800MHz+ 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+.

VII. CONCLUSION

This paper presents a new type of FPGA-optimised uniform random number generator, called a LUT-SR RNG. 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.

ACKNOWLEDGMENT

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.

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. FPGA*, 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] 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.
[5] M. Matsumoto and Y. Kurita, "Twisted GFSR generators II," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 3, pp. 254–266, 1994.
[6] I. L. Dalal and D. Stefan, "A hardware framework for the fast generation of multiple long-period random number streams," in *Proc. FPGA*, 2008, pp. 245–254.
[7] V. Sriram and D. Kearney, "A high throughput area time efficient pseudo uniform random number generator based on the TT800 algorithm," in *Proc. FPL*, 2007, pp. 529–532.
[8] 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. FPL*. IEEE Computer Society Press, 2005, pp. 275–280.
[9] P. L'Ecuyer and R. Simard, "TestU01 random number test suite," www.iro.umontreal.ca/~simardr/indexe.html, 2007.
[10] 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.