High Quality Uniform Random Number Generation Through LUT Optimised Linear Recurrences

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

Abstract

This paper describes a class of FPGA-specific uniform random number generators with a $2^k - 1$ length period, which can provide k random bits per-cycle for the cost of k Lookup Tables (LUTs) and k flip-flops. The generator is based on a binary linear recurrence, but with a recurrence matrix optimised for LUT based architectures. It avoids many of the problems and inefficiencies associated with LFSRs and Tausworthe generators, while retaining the ability to efficiently skip ahead in the sequence. In particular we show that this class of generators produce the highest sample rate for a given area compared to LFSR and Tausworthe generators. The statistical quality of this type of generators is very good, and can be used to create small and fast generators with long periods which pass all common empirical tests, such as Diehard, Crush, Big-Crush and the NIST cryptographic tests.

1. Introduction

Many applications are reliant on uniform random numbers, such as monte-carlo integration, simulated annealing, and financial simulations. Such applications require huge amounts of processing power, while offering plenty of scope to exploit fine-grain and coarse-grain parallelism, and so are often ideally suited to implementation in FPGAs. In order to function correctly, these applications require many parallel streams of high quality, large period, uncorrelated random number generators, and efficient hardware implementations offer an attractive solution. However, existing methods such as LFSR, Tausworthe and Cellular Automata based generators cannot provide all of these features at once.

In this paper we introduce a class of random number generators where every bit of the state is equally random, allowing large numbers of parallel number streams to be produced from one large period generator. The key contributions are:

- a technique for creating linear recurrence based random number generators optimised for LUT based architectures, particularly suited for applications where many random bits are needed per-cycle
- hardware implementation and benchmarking of the generators in the Virtex-II architecture
- statistical evaluation of the generator quality using the Diehard, Crush and NIST test batteries
- a comparison of the generators with other types of linear recurrence, such as LFSR and Tausworth based generators
- a combined generator which passes all empirical tests, with low area requirements and high generation speed

2. Background

Random number streams can be generators using either a True Random Number Generator (TRNG), or a Pseudo-Random Number Generator (PRNG). TRNGS rely on physical processes such as thermal noise or jitter, and so produce data that are fundamentally unpredictable. FPGA based implementations of TRNGs are available, such as [4] and [15], both variants on the same technique of sampling a high frequency clock with a low frequency unstable clock. While excellent for cryptographic purposes, these generators are generally not useful for simulations, as the bit generation rate is rather low, and because it is impossible to repeat a sequence without storing it.

Pseudo-Random Number Generators produce random numbers by using a deterministic function to transform the current state into a new state. The sequence of states produced is then used to form a sequence of random numbers. Because there are a finite number of states that can be produced, and the transition function is deterministic, the maximum sequence that any PRNG with k-bit state can produce is limited to 2^k. Selection of the state-transition function is obviously critical: $x_i = (x_{i-1} + 1) \bmod 2^k$ will produce a full length sequence, but is obviously not random. A good overview of common random number generators is available [7], but only random number generators appropriate for hardware implementation will be considered here.

0-7803-9407-0/05/$20.00 © 2005 IEEE

The two most common types of hardware random number generators are Linear Feedback Shift Registers (LFSRs) and Tausworthe generators, both based on binary linear recurrences, and Cellular Automata (CA) generators. Other algorithms are used for specialised tasks, such as the Blum Blum Shub algorithm [15] for cryptographic random numbers, but are not considered here.

Binary linear recurrence based generators work by forming each new bit in the state from a linear combination of the bits in the previous state. The advantage of this type of generator is that the state-transition function is easily and efficiently implemented in LUTs, state x_{i+n} can be determined from state x_i in $O(log_2(n))$ steps, and that the period length is only one less than the theoretical maximum. However, current generators from this family suffer from poor statistical quality. This type of generator is thoroughly discussed in section 3.

Cellular Automata generators form a large class of algorithms, including linear recurrences, but are usually taken to mean binary non-linear recurrences [16]. For example the well-known Rule-30 generator forms each new bit from a combination of the three nearest bits in the previous state according to the formula: $x_{i+1,b} = x_{i,b-1} \vee (x_{i,b} \oplus x_{i,b+1})$. This type of generator gives a chaotic sequence, i.e. the only way to find state x_{i+n} from x_n is to step through all the intermediate states. The period of a given generator is also difficult to determine, as there are likely to be multiple state-cycles of different lengths, with the initial state selecting which cycle is used.

One dimensional, nearest-neighbour CA generators have been used instead of LFSRs in VLSI for random bit generation [6], but the quality of the sequences is often lacking. In [13] more complex configurations are considered, such as four input functions to take advantage of 4-LUTs, and different connection topologies. This gives much higher statistical quality, but because by all four LUT inputs are used there is no easy way to load or store the generators state without partial reconfiguration or extra LUTs.

The quality of random number generators is usually determined through the use of empirical tests for sequence randomness. These operate on the sequence of numbers produced by a generator, rather than the generator algorithm itself. Each test looks for specific patterns within the sequence, then calculates the likelihood of that type of pattern occurring; for example, in the infinite limit a truly random bit sequence should consist of half zeroes, and half ones. Unfortunately it is only possible to test a finite number of samples, so the number of zeros is expected to follow a binomial distribution. By counting the number of zeroes found in a sample of numbers, then plugging this observed value into the inverse CDF (Cumulative Distribution Function) of the expected distribution, in this case a binomial CDF, a value between 0 and 1 is produced, often called a p-value. If a generator produces random numbers that pass the test, i.e. they fit that test's particular view of what is important in a random sequence, then the set of p-values from multiple runs of the test should be uniformly distributed. If the p-values are clustered around 0 or 1 then the generator does not meet that test's expectations about randomness. It is important to note that empirical testing is inherently probabilistic: a perfect random number generator will occasionally produce p-values that appear to indicate a failure.

Each empirical test only looks at one aspect of randomness, so it is common to group together lots of different tests into a test battery. The best known of these is Diehard [11], which comprises 16 different tests, and has been the standard test battery in recent years. Unfortunately Diehard is not parameterisable, and consumes just 2.5M 32-bit integers across all the tests; a hardware simulation running at 133MHz will consume over 50 times the Diehard sample size each second. TestU01 [9] is a newer test suite designed for modern applications that consume many more numbers. The standard test battery of the suite, Crush, consumes approximately 2^{35} numbers, while BigCrush, designed to test random numbers for long running applications, consumes 2^{38}. Another common test is the NIST test battery, which is designed to test random numbers for cryptographic purposes, and so has an emphasis on the ability to predict the next number from the previously generated ones.

3. Linear Recurrence Generators

In this section some of the theory behind linear recurrences for random number generation will be explained, along with the way that existing generators fit into this model.

A large family of software and hardware uniform random number generators, such as LFSRs and Combined Tausworthe generators, are based on linear recurrences using GF(2) (i.e. modulo 2) arithmetic. In their most general form these generators consist of a $k \times k$ matrix A, used to provide a sequence $x_1 ... x_{\inf}$ from an initial state x_0 using the recurrences:

$$x_n = Ax_{n-1}, \quad y_n = Bx_n \tag{1}$$

The k bit wide sequence is reduced down to a w bit wide output sequence using a $w \times k$ matrix B: This sequence can then be interpreted as a sequence of random numbers, most commonly by transforming to real numbers in the range $[0,1)$, or by interpreting as integers in the range $[0,2^w-1]$.

The parameter k is the number of state bits used by the generator, and ultimately determines the maximum period that can be provided. For a given matrix A there may be

multiple distinct sequences that can be entered, depending on the initial value x_0. The maximum period achievable is $p = 2^k - 1$, starting from $x_0 \neq 0$. It is impossible to achieve a sequence of length 2^k, as there is no way to create a matrix A that will transform a vector of zero to anything other than zero under GF(2), so the best that can be achieved is one cycle of length 1 and another of length $2^k - 1$.

The condition for maximum period is that the recurrence matrix must have a characteristic polynomial which is primitive modulo 2. The characteristic polynomial is defined as $P(z) = det(A - Iz)$, so for a $k \times k$ matrix this will be a polynomial of degree less than or equal to k. The sequence generated by A has maximum period if and only if $P(z)$ is primitive modulo 2 [10].

Parameter w determines the number of output bits provided by the generator, and the matrix B is used to determine how the output bits are created from the state bits. If $B = I$ then the state bits will be used directly, but if $B \neq I$ then the output bits will comprise some linear combination of the state bits. This process is often called tempering, and can be used to improve the statistical properties of the output sequence, for example by using two state bits to provide each output bit when $k \geq 2w$.

The two matrices A and B are chosen to provide an output sequence that is of high statistical quality, while also being easy to implement. Ease of implementation breaks down into two further categories, of software and hardware: In software it is necessary that the matrix multiplications can be implemented efficiently using full-length word operations, while in hardware it is desirable to minimise the amount of logic and registers used. Satisfying any two of these three conditions often means that the third one is not met; for example generators that can be easily implemented in software and have good statistical quality often require too much state to be implemented in hardware.

The classic hardware random number generator is the single bit LFSR. This generator is based on very simple maximum period linear recurrences, by selecting a primitive polynomial of the appropriate degree, then setting up a simple recurrence that implements the polynomial directly. This is usually generated as a bit sequence, $b_{i+1} = w_i b_{i-1} + w_{i-1} b_{i-2} ... w_{i-k} b_{i-k}$, where $w_1 ... w_k$ are the coefficients of the polynomial. The generators obviously still has a k bit state, formed from the last k bits, $x_i = < b_i, b_{i-1}, ..., bi - k >$, but because most of the state is just a shifted version of the previous state only 1 bit can be used. LFSRs have very efficient implementations in certain architectures [5], but because each instance only produces 1 bit per cycle, w parallel instances are needed to produced a w bit number sequence. So to produce a $2^k - 1$ bit sequence, kw bits of state are needed, rather than just k. LFSRs also become less area-efficient as the number of taps increase and the period length is increased, so are not appropriate for high quality random number (as opposed to bit) generators.

The Tausworthe generator [8] is a type of generator that avoids some of the problems with parallel LFSRs, in particular all bits of the state can be used. A Tausworthe sequence is created by taking w bit blocks from a maximum period k bit recurrence ($k \leq w$) every s bits, i.e. $x_i = < b_{is+1}, b_{is+2}, ..., b_{is+w} >$. If $2^k - 1$ and s are relatively prime then the overall period of the sequence x will remain $2^k - 1$. It may appear that each state transition will require s steps, but it is possible to calculate each transition in parallel; for example the QuickTaus algorithm [8] can be used in both software and hardware to implement Tausworthe generators for primitive trinomials.

Because Tausworthe generators are usually implemented using trinomials, the quality of the generators is rather poor, particularly when $s < w$. The main use of the Tausworthe generator is to create Combined Tausworthe generators, whereby two or more w bit wide generators are combined using $\oplus$ to provide a new sequence. If the constituent polynomials are chosen such that all the periods are relatively prime, then the product of the overall sequence will be relatively prime. Although implemented as a combination of three separate generators, the overall combination forms another recurrence, though with a non-maximum period sequence. Combined Tausworthe Generators are area efficient, and can produce good quality generators. One drawback is that the maximum period that can be achieved for a given w is limited, as the maximum degree that can be used is w, but all the other polynomials must be smaller yet still coprime. Also the period does not meet the maximum possible for the number of state bits, although it is quite close.

4. LUT Optimised Linear Recurrences

The Tausworthe generator is primarily designed for software use, with low instruction count implementation being the major priority. The left side of figure 1 shows the recurrence matrix for a 31-bit Tausworthe generator, which takes six instructions to execute in software. In hardware this will take 31 FFs and 22 4-LUTs, and only two inputs of each LUT entry will be used. This is a waste of logic as only half the LUT's processing power will be used.

If software implementations are completely ignored, then designing the generator recurrence matrix becomes much simpler: to achieve maximum period all bits must depend on at least one other bit, and must be used by at least one other bit; if a bit is to appear random, rather than just a shifted copy of another bit, then it must depend on at least two bits; a 2 input function requires one l-LUT, but the extra $l - 2$ inputs may as well be used as it costs nothing; all bits should only be sampled by l other bits to avoid

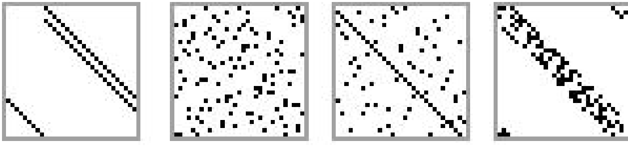

Figure 1. Feedback matrices for, from left to right: 31-bit Tausworthe generator, 4-tap matrix, 3-tap loadable matrix, 4-tap ring matrix.

over-dependence on some bits within the state; the matrix must have maximum-period. In matrix terms this means that all rows of the matrix must contain l ones, all columns of the matrix must contain l ones, and the characteristic polynomial must be primitive.

To find such matrices a stochastic search approach is used, which generates random candidate matrices, then applies progressively stricter tests for maximum period to each one:

1. Generate a $k \times k$ matrix A.
2. If $det(A) = 0$ then go to step 1.
3. Generate $P(z)$ from matrix A.
4. If $P(z)$ is reducible then go to step 1.
5. If $P(z)$ is primitive then return matrix A.

Performing a full primitivity test is computationally expensive, so it is important to reject matrices as quickly as possible. Step 2 immediately rejects many matrices without even having to calculate $P(z)$, a relatively expensive step in itself. Step 4 rejects yet more matrices without needing a full primitivity test. Only a small number of matrices make it to step 5, and a relatively high proportion of those actually are primitive.

This search process is implemented using the NTL Number Theory Library [14] for matrix storage and manipulation, and the calculations in steps 2, 3 and 4. The final primitivity test is performed by a version of PPSearch, modified to accept NTL format GF(2) polynomials. This system can be used to find full period matrices up to a size of about 1500, but beyond this point a more efficient algorithm, or hardware accelerated implementation, will be needed.

Table 1 shows some statistics from the search process while searching for matrices with $l = 4$ for increasing matrix size. For each size four different matrices are found, and the table shows the aggregate statistics. The *Tested Candidates* figure is the total number of candidate matrices tested, while the *Rejections* columns show how many matrices are rejected by each stage. A very small proportion of non-primitive matrices make it through to the primitivity test, with most being rejected by the Determinant test. The *Total time* column is the total CPU time used to find the four generators, measured on an Athlon 1.2GHz machine with 1GB of RAM. Also included is a breakdown of where the time is spent, and it is clear that by far the biggest bottleneck is the characteristic polynomial generation, which is slowly coming to dominate the entire process.

After implementing the search process, it was discovered that the requirements outlined above, that each row and column must have exactly l ones, never produces any full-period generators. The solution found is to select one or two bits in the state and either use an $l+1$ input feedback, or an $l-1$ input feedback. Only one modified bit seems to be necessary in order to find a solution, but scaling the number up with the matrix size speeds up the search process. The first solution requires an extra LUT for the selected bits, while the second solution possibly sacrifices a little quality. In this paper the second solution is used, but where possible the $l-1$ input bit(s) are not directly used to form random numbers, hopefully hiding this minor flaw.

The right hand side of figure 1 shows a 31 bit recurrence matrix generated for a 4-LUT architecture. The difference from the Tausworthe generator to the left is visually clear, and in section 7 the statistical quality will also be evaluated, but first some alternate matrix constraints will be considered that organise the feedback in different ways.

The first modification is to allow the generator's state to be read and stored, which is necessary in order to be able to start the sequence from a specific state. This is particularly important in parallel simulations, as each generator needs to be initialised in a specific state in order to make sure that there is no overlap between the random sequences produced during the run. This is a problem if all l inputs of the LUT are already used, as two extra inputs are needed for each bit in the state: one to control whether the bit will be formed from a recurrence or loaded from an external source, and another to supply the bit from an external source. Implementing this function will require two LUTs, one to implement the original recurrence, and another to select between the recurrence input and the external input on the basis of a control input.

One option is to increase the number of feedback taps from l to $2l-3$ by using two LUTs, increasing the complexity of the recurrence as well as supporting loading. If doubling the number of LUTs is unacceptable, then state loading can be implemented with just one input: the control signal. This is achieved by loading the state serially in k cycles, rather than concurrently in a single cycle. A k bit cycle through the state bits is chosen from the set of connections already used to form a matrix with $l-1$ inputs per bit. This cycle of bits forms a shift register, which is used to load new state bits in serial. The control bit uses up the final input in each LUT, and selects between just using the single connection shift register connection to load a new state, or all of the connections to calculate the next state.

Matrix size	Tested candidates	Rejections			Total time (s)	Percentage of total time				
		Det	Irred	Prim		Generate	Det	CharPoly	Irred	Prim
32	591	417	169	1	0.46	15.4%	12.3%	56.1%	9.7%	6.4%
64	1619	1151	461	3	6.09	6.8%	7.1%	77.9%	7.1%	1.0%
128	5570	3964	1601	1	145.07	2.6%	3.4%	89.0%	4.9%	0.1%
192	3898	2812	1076	6	332.10	1.6%	2.2%	92.1%	4.0%	0.1%

Table 1. Search process statistics for finding primitive 4-LUT generators with increasing matrix size.

In a 4-LUT architecture, such as the Virtex [3] family, this arrangement will reduce each bit's state transition to a linear combination of three other bits. This lack of feedback complexity can be compensated for by organising the feedback matrix such that the w bits used to form an output stream only depend on the other $k-w$. This avoids the simplest correlations between bits within the output stream, and can be extended for multiple streams taken from the same generator.

In other architectures this arrangement can be implemented with no overhead. For example, the Stratix II device [1] adopts a flexible LUT architecture, and one of the modes allows two 5-LUTs per cell, as long as two of the inputs are common to both LUTs. This configuration can be used to implement a 4-input per bit recurrence generator with serial state loading, as one of the shared inputs will be used by the control signal, while the other can be found simply by grouping together pairs of bits that already depend on a common input.

Another constraint on matrix generation is to try to reduce routing congestion, by only allowing bits to sample other bits within t bits of itself. Figure 1 shows a 128 bit matrix where such a constraint with $t = 8$ has been used. When implemented in hardware this form of matrix would be expected to form a ring of registers with only local connections and so be able to achieve higher speeds than a more general matrix. Finding matrices with low values of t takes a long time, with $t = k/8$ being a reasonable lower point for the current search process. It was also found that the place-and-route tools actually produced slower designs for all ring-based matrices that were tried, so this organisation is not considered in the evaluation section.

5. Implementation

In this section the hardware performance of the generators is tested using Handel-C implementations in the Virtex-II architecture.

Given a binary recurrence matrix it is extremely easy to create a hardware description that implements it. For example, the following program segment:

```
macro expr size = k ;
macro expr matrix = {
  { 0,1,...0,0},...,{1,0,...1,0}
};
macro expr mkFB(i,row)=select(i==size,0,
    (state[i]&row[i])^mkFB(i-1,row);
bool state[size];
par(i=0;i<size;i++){
  state[i]=buildFB(0, matrix[i]);
}
```

is sufficient to implement a basic generator in Handel-C. The elements of the recurrence matrix are inserted into the constant array *matrix*, and then the recurrence is directly evaluated. In practice it is more efficient to generate the source code per matrix, with the feedbacks explicitly encoded in the source code. This is implemented as a function of the matrix search program, allowing Handel-C source code to be generated directly from the matrix. Two types of hardware can be generated: one that implements just the generator core for area and speed measurements, and another that also contains interfacing code to software for statistical testing.

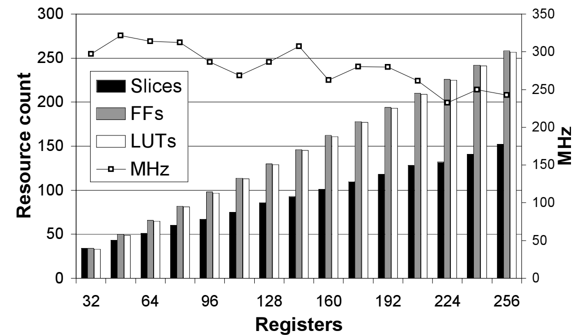

Figure 2. Changes in area and speed for 4-tap matrix generators on the Virtex-II architecture.

Figure 2 shows the area and speed of a set of 4-input matrices for increasing matrix size. Both FF (Flip-flops) and LUT (Lookup Tables) counts are exactly linear, with a k size matrix requiring $k+1$ LUTs, and $k+2$ FFs. These relationships are exactly as hoped, showing that the attempt to target the LUT architecture has worked.

This same relationship between area and k is seen in the loadable and ring matrices. The only minor difference is the loadable matrix, where two extra FFs are used: one

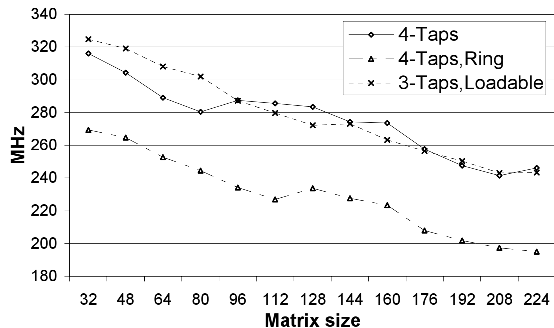

Figure 3. Comparison of generator speed for different matrix types on the Virtex-II architecture.

to buffer the control signal, and another to buffer the serial data input. It is possible that once embedded in a real design the area taken up by the generators would increase slightly due to replication. For example the control signal might be replicated to improve timing, or feedbacks for certain bits might be calculated twice, once to supply the feedback, and once to allow the random bit to be used in a different area of the circuit.

The speed of the generator for increasing k is shown in figure 3, as well as the speeds for the two other types of matrix. As would be expected, the speed gradually decreases as the size of the matrix increases due to longer connections and routing congestion. The speeds of two of the matrix types begin to converge, *except* for the matrix type that is explicitly designed for better speed. This is probably due to the placer using a random initial placement of the state FFs, and then never managing to rearrange them so that bits are close to their neighbourhood of bits in the ring.

6. Further Optimisations

As show in section 7, the statistical quality of the generators shown so far is good, but suffers from the same problem as any linear recurrence based generator: the next state of a linear recurrence based generator can always be predicted if more than k previous states are known. This is why none of the given generators pass the linear complexity statistical tests. Here we will outline one modification that can be used to pass these tests, while still retaining all the good properties of recurrence generators, such as low area, high speed, and the ability to skip the sequence ahead.

Increasing the value of k until each test passes treats the symptoms, but not the underlying problem. A better solution is to combine two samples using addition or multiplication. The underlying linear recurrence is then masked due to the mixing of bits. Multiplication does the best job of mixing, but requires high-cost resources in hardware, so here addition is chosen. One problem with combining through addition is that the lowest bit is simply the exclusive-or of the least significant bits of the inputs. To make sure that even the low output bit is of good quality, the lowest d bits produced by the addition will be discarded, so to produce a w bit output a $w+d$ bit adder is used. If w is large, e.g. 32 bits, then this adder is likely to limit clock speed, so instead the addition is split up into s separate additions of $w/s+d$. To supply this addition a total of $w+sd$ random bits are needed to produce each output sample.

This additive combination scheme is implemented using $w=32$, $s=4$, and $d=2$. The two input samples are supplied by two separate 3-tap matrix generators, one of size 80, the other 81, both generated to support for serial loading. Because the periods of the two generators are coprime the full period will is $(2^{80}-1)(2^{81}-1)$ giving a period of approximately 2^{160}. Two separate generators are used rather than one single generator, as it should improve speed in congested designs. This generator can produce a single stream, or by using two additive combination stages, two streams. Higher period generators that support more streams can easily be created by using larger matrices, and different width streams can also be generated from a single generator if necessary.

As well as passing the Diehard and Crush tests, this generator also passes the harder Big-Crush test. The NIST test for cryptographic numbers is also passed, using a 1Gb sample treated as 1000 independent streams. When two streams are generated, both pass all the tests, and so far no empirical test batteries have been found that it does not pass.

7. Evaluation

Testing randomness with a test battery, such as Diehard, does not provide a definite answer to the question of whether a given sequence is random or not. All the tests provide is a set of p-values which must then be interpreted. One approach to this is to run the tests, and consider any values outside the $[0.01, 0.09]$ range as a fail, but in a set of 100 p-values at least one value in this range should "fail".

The approach taken here is to run each test battery three times, and then for each test within the battery the triple of corresponding p-values are considered. Tests are considered a fail if one of three conditions hold: at least one p-value outside the range $[0.0001, 0.9999]$; at least two p-values outside the range $[0.01, 0.99]$; or all three p-values outside the range $[0.05, 0.95]$. This means that there is very roughly a 1 in 10000 chance that the wrong decision is made.

The tests are performed by executing the matrix generators in hardware using an RC2000 system [2] which contains an XC2V6000 FPGA, with a software wrapper to return the generated samples back to the test suites. The gen-

erators are initialised to a random state before each test, and strictly consecutive samples are returned to the test suite, i.e. no samples are dropped or skipped.

A feature of the matrix generators is that all k bits are usable, so the quality of all k/w streams of some of the generators were tested. It is found that the streams are all of roughly the same quality, and in only one case (where $k = 256$) is the quality of one stream significantly worse than another. In that case the stream is supplied by a set of bits with very low connectivity to the rest of the matrix, forming an almost independent stream.

Table 2 shows these results under the Virtex-II column, listing the number of failed tests found when applying the Diehard and Crush tests. The first group of results shows a selection of 4-tap (i.e. non-loadable) generators, while the second group shows 3-tap generators that support serial state loading. The third group shows the additive combination generator from section 6, first where just one 32 bit stream is produced, then where two streams are produced. The fourth group contains other hardware generators for comparison purposes, while the last group contains results from some software generators running on a 3.2GHz P4, including the Mersenne (mt19937) [12].

Looking at the Diehard results, the slight loss in randomness in the 3-tap generators is clear, as the 4-tap generators pass with at $k = 96$, while the 3-tap generators only pass at 128. The Crush results show this as well, with the 4-tap generators passing more tests for the same k value. The parallel LFSR generator gives similar quality at $k = 64$, but requires about 3 times the area, even with the SRL16 optimisations used by CoreGen, and when the LFSR period is doubled, the quality does not improve by much.

The Tausworthe generators provide much better quality than the LFSRs, and are actually better than the matrix generators for a similar period length; this is not too surprising, as the generators in [8] are selected to give Maximal Equidistribution and so are in one sense optimal, while the matrices tested here are chosen essentially at random, with maximum period as the only criteria. A search for matrices with good equidistribution should provide results at least as good as the Tausworthe for the same period. For larger periods the matrix generators achieve equal or better quality, while requiring less logic per sample generated: the *4-tap,k* $= 256$ (table 2) generator is of about the same quality as Taus113, but has six times the pure sample rate, and achieves 4.3 times the sample rate per LUT used.

When high quality number generation is considered, the LFSR based generators cannot compete due to large area and poor quality. For instance, the *combo,2-stream* generator produces over three times the sample rate per LUT compared to *LFSR-160*, and has much better quality. The Taus113 generator requires a relatively low amount of area, but still does not pass all the tests, while the dual combination generator has roughly the same sample generation rate per LUT, and is of much higher quality. These generators also perform well in the Spartan-3 architecture, operating at about 85% of the Virtex-II speed.

Two of the Crush tests are not passed by any of the basic matrix generators, or by the LFSR and Tausworthe generators. These are two tests for linear complexity, and so easily detect the linear structure of the relatively low period generators shown here. Another two tests are only passed by the two matrix generators with $k = 1248$, which are both tests for matrix rank. These tests can detect linear recurrences below a certain degree, in the case of Crush the maximum degree is 1200. For evaluation purposes a period just over 1200 is chosen, just to check that it could be passed. A better solution is the modifications suggested in section 6.

8. Conclusion

In this paper a novel technique for designing and implementing linear recurrence based generators in LUT based architectures has been demonstrated. By designing the recurrence matrix to make maximum use of LUT inputs, it is possible to make high quality random number generators with relatively few resources. A generator with period $2^k - 1$ can be implemented using just k Flip-Flops and k LUTs. All k bits of the state are random, allowing multiple streams of numbers to be sourced from a single generator, rather than requiring one generator per random number stream.

Table 2 summarises the statistics for some of the suggested generators, as well as the Taus113 and the software Mersenne Twister. The LUT optimised generators can offer high period and very high speed sample generation for a modest area cost, particularly when multiple streams are taken from one generator.

By combining two of these generators, it is possible to create an FPGA 32-bit random number generator with a period of 2^{160} that passes all common empirical tests, including Crush, Big-Crush and the NIST suite, for a cost of just 307 Flip-Flops and 202 LUTS, running at a speed of 210MHz in the Virtex-II architecture (combo,1-stream design in table 2). This type of generator is ideal for parallel simulations, as the generator state can be read and written at runtime, and the generator state at arbitrary points in the future can be efficiently calculated.

There are several avenues for further work. Different full-period matrices found using the same constraints often have very different statistical quality, so it would be useful to examine large numbers of matrices found using the same constraints to try to determine some upper bounds for quality. The empirical tests can also be supplemented by a search for matrices with good Equidistribution [8], a theoretically derived property which is a good indicator of

Generator	Period (log_2)	Diehard Failed Tests	Crush Failed Tests	FFs	LUTs	Virtex-II		Spartan-3	
						MHz	Gb/s	MHz	Gb/s
4-tap,k=32	32	3	14	34	33	309	9.9	284	9.1
4-tap,k=64	64	2	12	66	65	310	19.8	271	17.3
4-tap,k=96	96	0	10	98	97	289	27.7	259	24.9
4-tap,k=128	128	0	7	130	127	287	36.7	236	30.2
4-tap,k=256	256	0	6	258	257	246	63.0	200	51.2
4-tap,k=1248	1248	0	2	1250	1249	168	209.7	132	164.7
3-tap,k=32	32	5	17	36	33	302	9.7	298	9.5
3-tap,k=64	64	2	13	68	65	319	20.4	286	18.3
3-tap,k=96	96	1	11	100	97	308	29.6	268	25.7
3-tap,k=128	128	0	8	132	127	287	36.7	238	30.5
3-tap,k=256	256	0	6	260	257	243	62.2	214	54.8
3-tap,k=1248	1248	0	2	1252	1249	173	215.9	152	189.7
combo,1-stream	160	0	0	307	202	210	6.7	181	5.8
combo,2-stream	160	0	0	387	242	207	13.2	176	11.2
Taus88	88	1	9	132	129	304	9.7	277	8.9
Taus113	113	0	6	164	161	287	9.1	258	8.3
LFSR-64	64	3	15	291	321	272	9.0	232	7.4
LFSR-160	160	2	14	451	481	252	8.1	209	6.7
Taus88 (SW)	88	1	9			106.6	3.4		
Taus113 (SW)	113	0	6			81.1	2.6		
Mt19937 (SW)	19937	0	0			63.7	2.0		

Table 2. Summary of the quality, area and speed of a selection of hardware generators.

randomness.

Different FPGA families offer opportunities for increasing quality or reducing area using architecture specific components. The Virtex SRL16 could be used to provide high periods when not all bits of the state will be consumed, while the Apex II flexible LUT architecture offers the possibility of prioritising the quality of some bits, by using higher input count LUTs for those bits.

References

[1] Altera. http://www.altera.com.

[2] RC2000 accelerator card. http://www.alpha-data.com/adm-xrc-ii.html.

[3] Xilinx. http://www.xilinx.com.

[4] Viktor Fischer and Milos Drutarovský. True random number generator embedded in reconfigurable hardware. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 415–430, London, UK, 2003. Springer-Verlag.

[5] Maria George and Peter Alfke. Linear feedback shift registers in virtex devices. Technical report, Xilinx, 2001.

[6] P. D. Hortensius, R. D. McLeod, and H. C. Card. Parallel random number generation for vlsi systems using cellular automata. *IEEE Trans. Comput.*, 38(10):1466–1473, 1989.

[7] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 81.

[8] Pierre L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Mathematics of Computation*, 65(213):203–213, 1996.

[9] Pierre L'Ecuyer and Richard Simard. TestU01. http://www.iro.umontreal.ca/ simardr/indexe.html.

[10] G. A. Marsaglia and L.H. Tsay. Matrices and the structure of random number sequences. *Linear Algebra Appl*, 67:147–156, 1985.

[11] George Marsaglia. The diehard random number test suite. `http://stat.fsu.edu/pub/diehard/`.

[12] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[13] Barry Shackleford, Motoo Tanaka, Richard J. Carter, and Greg Snider. FPGA implementation of neighborhood-of-four cellular automata random number generators. *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, 2002.

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

[15] K. H. Tsoi, K. H. Leung, and P. H. W. Leong. Compact FPGA-based true and pseudo random number generators. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 51. IEEE Computer Society, 2003.

[16] Stephen Wolfram. Random sequence generation by cellular automata. *Adv. Appl. Math.*, 7(2):123–169, 1986.