Journal of VLSI Signal Processing 47, 77–92, 2007

DOI: 10.1007/s11265-006-0014-9

High Quality Uniform Random Number Generation Using LUT Optimised State-transition Matrices

DAVID B. THOMAS AND WAYNE LUK
Department of Computing, Imperial College London, South Kensington Campus, London, UK

Received: 6 February 2006; Accepted: 13 November 2006

Abstract. This paper presents a family of uniform random number generators designed for efficient implementation in Lookup table (LUT) based FPGA architectures. A generator with a period of 2^k-1 can be implemented using k flip-flops and k LUTs, and provides k random output bits each cycle. Each generator is based on a binary linear recurrence, with a state-transition matrix designed to make best use of all available LUT inputs in a given FPGA architecture, and to ensure that the critical path between all registers is a single LUT. This class of generator provides a higher sample rate per area than LFSR and Combined Tausworthe generators, and operates at similar or higher clock-rates. The statistical quality of the generators increases with k, and can be used to pass all common empirical tests such as Diehard, Crush and the NIST cryptographic test suite. Theoretical properties such as global equidistribution can also be calculated, and best and average case statistics shown. Due to the large number of random bits generated per cycle these generators can be used as a basis for generators with even higher statistical quality, and an example involving combination through addition is demonstrated.

Keywords: Uniform Random Numbers, FPGA, Simulation

1. Introduction

Many applications are reliant on random numbers, such as financial calculations, simulated equipment testbeds, and simulation of communications channels. Such applications require large amounts of processing power, while providing many opportunities to exploit fine-grain and coarse-grain parallelism, and so are often ideally suited to implementation in FPGAs [5, 20, 30]. In order to function correctly, these applications require many parallel streams of high quality, large period, uncorrelated uniform random number generators. These are most commonly used as input to transformation functions which will provide the non-uniform distributions, and typically require many uniform input bits for each non-uniform output sample [4, 15].

In this paper we introduce a class of random number generators where every bit of the generator state can be used as a random output bit, 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, using state-transition matrices optimised for LUT based architectures; it is particularly suited for applications where many random bits are needed per-cycle
- Hardware implementation and benchmarking of the generators in the Virtex-II architecture
- An example of an additively combined generator which passes all empirical tests, with low area requirements and high generation speed

– Empirical evaluation of generator quality using the Diehard, Crush and NIST test batteries, and theoretical evaluation using the equidistribution test
– A comparison of the generators with other types of linear recurrence, such as LFSR and Combined Tausworth based generators

2. Background

Random number streams can be generated 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 [7] and [26], which are 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 too low, typically only tens or hundreds of kilobits per second. TRNGs also make it impossible to repeat a random sequence unless the entire sequence is stored, meaning that it is impossible to repeat a specific simulation run in order to verify results.

Pseudo-Random Number Generators produce random numbers by using a deterministic state-transition function $f(x)$ to transform the current state x_i into a new state x_{i+1}. The sequence of states $x_1, x_2, \ldots$ is then used as 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 length that any PRNG with k-bit state can produce is limited to 2^k. Selection of the state-transition function is obviously critical: $x_{i+1} = (x_i + 1) \bmod 2^k$ will produce a full length sequence, but is obviously not random. A good overview of common random number generators is provided by Knuth [11], but concentrates mainly on software generators. Here a brief survey of techniques appropriate for hardware implementation is presented.

The two most common types of hardware random number generators are Linear Feedback Shift Registers (LFSRs) and their variants, and Cellular Automata (CA) generators. Other algorithms are also used for more specialised situations, such as the Blum Blum Shub algorithm [26] for cryptographic random numbers, but are not appropriate for situations requiring high sample rates such as simulations.

LFSRs are the best known of a family of generators based on binary linear recurrences [25], that includes other generators used in hardware such as Tausworthe, Combined Tausworthe, as well as the new family of generators introduced in this paper. Some software generators such as the Mersenne Twister [19] and WELL [22] also belong to this family, but are less commonly implemented in hardware.

Binary linear recurrence based generators form each new bit in the next state from a linear combination of the bits in the current 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 discussed in more detail 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 [27]. 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: $x_{i+1,b} = x_{i,b-1} \oplus (x_{i,b} \vee x_{i,b+1})$. An example of a 6-bit CA is shown in Fig. 1, where each register bit is updated using a combinatorial function of its current state plus that of its two neighbours. The array of bits is typically organised as a ring, so the first and last bits are considered to be neighbours. 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

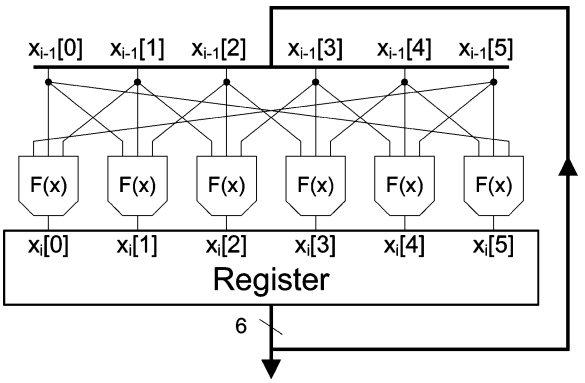

Figure 1. Six bit one-dimensional Cellular Automata (*CA*) circuit.

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 [10], but the quality of simple one-dimensional sequences is often poor. In [23] more complex configurations are considered, such as four input functions to take advantage of 4-LUTs, and different connection topologies. This gives higher statistical quality, but because all four LUT inputs are used there is no easy way to load or store the generator's state without partial reconfiguration or extra LUTs. The generators also still have unsolved problems related to sequence period and quality, due to the lack of formal methods for analysing CAs.

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 [16], which comprises 16 different tests, and has been the standard test battery in recent years. Unfortunately Diehard is not parametrisable, and consumes just 2.5 M 32-bit integers across all the tests; a hardware simulation running at 133 MHz will consume over 50 times the Diehard sample size each second. TestU01 [13] 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 Big-Crush, 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 (although the test does not confer any guarantee of algorithmic cryptographic strength), 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 binary linear recurrences for random number generation will be introduced, 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 or binary) arithmetic. In their most general form these generators consist of a $k \times k$ matrix A, used to provide a sequence $x_1 \ldots x_{\text{inf}}$ from an initial state x_1 using the recurrences:

$$x_{i+1} = Ax_i, \quad y_i = Bx_i \tag{1}$$

The k bit wide sequence is reduced down to a w bit wide output sequence using a $w \times k$ matrix B. The sequence $y_1, y_2, \ldots$ 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_1. The maximum period achievable is $p = 2^k - 1$, starting from $x_1 \neq 0$. 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 all zero to anything

other than zeros under GF(2): the best that can be achieved is one cycle of length 1 when $x_1 = 0$, and another of length $2^k - 1$ when $x_1 \neq 0$.

The condition for maximum period is that the recurrence matrix must have a characteristic polynomial which is primitive modulo 2 [25]. 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 [17].

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 [18], 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 area-efficient in hardware.

The classic hardware linear recurrence based 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 recurrence that implements the polynomial directly. This is usually generated as a bit sequence, $b_{i+1} = w_1 b_i + w_2 b_{i-1} \ldots w_k b_{i-k+1}$, where $w_1 \ldots w_k$ are the coefficients of the polynomial. The generator obviously still has a k bit state, formed from the last k bits, $x_i =< b_i, b_{i-1}, \ldots, b_{i-k+1} >$, but because most of the state is just a shifted version of the previous state only 1 bit can be used as an output. Figure 2 gives an example of the structure of a 6-bit LFSR, based on the feedback polynomial $b^6 + b^5 + b^0$.

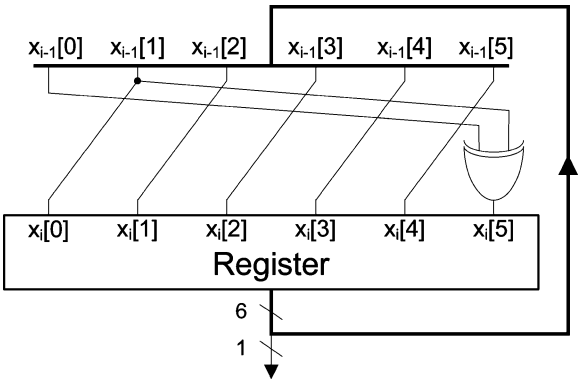

Figure 2. A 6 bit Linear Feedback Shift Register (*LFSR*).

LFSRs have very efficient implementations in certain architectures [1], particularly in the Virtex-II and later families from Xilinx, where the shift register portions can be implemented using SRL16s [9]. However, 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 or the period length is increased, so are not appropriate for high quality random word (as opposed to bit) generators.

The Tausworthe generator [12] is a type of linear recurrence generator that avoids the main drawback of LFSRs, as more than one bit from the state can be used as an output each cycle. A Tausworthe sequence is created by taking w bit blocks from a maximum period k bit recurrence ($w \leq k$) every s bits, i.e. $x_i =< b_{is+1}, b_{is+2}, \ldots, b_{is+w} >$. 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 [12] 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 poor, particularly when $s < w$. The main use of the Tausworthe generator is to create Combined Tausworthe generators [12], whereby two or more w bit wide generators are combined using exclusive-or to provide a new sequence. If the constituent polynomials are chosen such that all their periods are relatively prime, then the period of the combined

generator is equal to the product of the polynomial periods. Although implemented as a combination of three separate generators, the overall combination forms another linear recurrence matrix, though with a non-maximum period sequence. Combined Tausworthe Generators are area efficient (compared to parallel LFSRs), and produce good quality generators [14, 29].

4. LUT Optimised Linear Recurrences

The Tausworthe generator is primarily designed for software use, with a low instruction count implementation as the main design priority. The left side of Fig. 3 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 inputs will be used.

If the requirements of software implementations are ignored, then designing the generator recurrence matrix becomes much simpler. The restrictions imposed on the matrix to allow efficient calculation using word-based bit-wise instructions can be ignored, allowing the matrix to be designed for efficient LUT based implementation. The rules for selecting an efficient matrix are then as follows:

- A minimal criterion for maximum period is that all bits must depend on at least one other bit, and must in turn be used by at least one other bit.
- If a bit is to appear minimally random, rather than just a shifted copy of another bit from a previous state, 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.

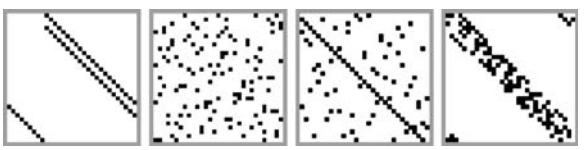

Figure 3. Feedback matrices for, from left to right: 31-bit Tausworthe generator, 4-tap matrix, 3-tap loadable matrix, 4-tap ring matrix.

- Ideally all bits should only be sampled by l other bits to avoid over-dependence on specific bits within the state.
- The matrix must have maximum-period, i.e. the characteristic polynomial of the matrix must be primitive (modulo 2).

These rules mean that a $k \times k$ matrix must be found, where all rows of the matrix contain l ones, all columns of the matrix contain l ones, and the characteristic polynomial is primitive (as explained later, the row and column constraints have to be relaxed slightly in practise).

To find such matrices a stochastic search approach is used, which generates random candidate matrices that satisfy the constraints on tap placement until a matrix is found which has a primitive characteristic polynomial. Both calculating the characteristic polynomial and testing a polynomial for primitivity are time-consuming processes, so some quick rejection steps are used. First, if the determinant of a binary matrix is zero then the characteristic polynomial cannot be primitive. Second, a necessary (but not sufficient) condition for polynomial primitivity is that the polynomial is irreducible. This leads to the following algorithm for finding generator matrices:

1. Generate a random $k \times k$ matrix A with approximately l ones in each row and l ones in each column.
2. If $\det(A) = 0$ then go to step 1.
3. Calculate $P(z)$, the characteristic polynomial of A.
4. If $P(z)$ is reducible then go to step 1. This step is performed using a fast probabilistic algorithm that will occasionally not reject a reducible matrix.
5. Perform full primitivity test on $P(z)$. If $P(z)$ is primitive then accept matrix A as a full-period generator. The primitivity test rejects the small number of reducible matrices that were misclassified in step 4 by the probabilistic test.

This search process is implemented using the NTL Number Theory Library [24] to implement the calculations in steps 2, 3 and 4. The final primitivity test is performed by a version of PPSearch [6], modified to accept NTL format binary polynomials. This system can be used to find full period matrices up to a size of about 1,500, but beyond this point a more efficient algorithm, or hardware accelerated implementation, will be needed.

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

		Rejections				Percentage of total time				
Matrix size	Tested candidates	Det	Irred	Prim	Total time (s)	Generate in %	Det in %	CharPoly in %	Irred in %	Prim in %
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 shows statistics from the search process while searching for matrices with $l = 4$ for increasing matrix size. For each size the search process is run until four different full-period matrices are found, and the table shows the overall 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 1.2 GHz machine with 1 GB 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 calculation, which increasingly dominates execution time as the matrix size increases.

After implementing the search process, it was discovered that the requirements outlined above, specifically that each row and column must have exactly l ones, results in matrices that are never full-period generators. The solution that is adopted is to select one or two bits in the state and either use an $l + 1$ input feedback or an $l - 1$ input feedback for those bits. 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.

Equation (2) shows an example of a six bit full-period generator, with $l = 3$. Note that the bottom row only contains two ones in order to allow the full-period criteria to be met. The equivalent generator circuit is also shown in Fig. 4.

$$x_{i+1} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} x_i \qquad (2)$$

The right hand side of Fig. 3 shows a larger 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

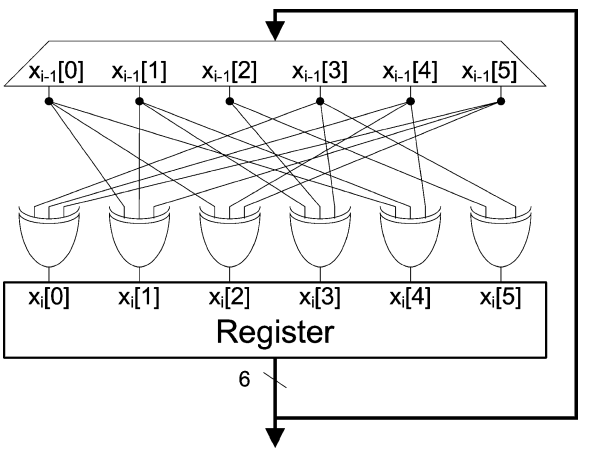

Figure 4. Six-bit 3-tap LUT optimised linear Recurrence.

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 simulation node needs to operate within a different sub-sequence of the generators entire sequence. This can be achieved by assigning starting states at known offsets within the sequence to each simulation node, using the property of linear recurrences that $x_{i+t} = A^t x_i$. For example, if each of N simulation nodes will consume t random outputs, each node $1 \leq n \leq N$ can be given a starting state $s_n = A^t s_{n-1}$, where s_0 is some arbitrary base state. However, this requires some way of loading arbitrary states into each generator.

Loading state into a generator is a problem if all l inputs of each 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. For example in a 4-LUT device this would increase the width of each exclusive-or to five inputs.

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 in parallel 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.

In a 4-LUT architecture, such as the Virtex [28] 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 [3] 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. Alternately the SLOAD feature can be used to implement the serial loading, but this may require device specific HDL.

The major factor that limits performance in this architecture is routing congestion: even in the simple six bit example shown in Fig. 2 the routing is already very complex. An attempt to reduce routing congestion was made, by restricting the matrix to only connect together bits within t bits of each other (when the state is considered as a ring of bits). Figure 3 shows a 31 bit matrix where such a constraint with $t = 5$ has been used, showing that all feedback taps are clustered along the main diagonal. 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. In practise it was found that such matrices were consistently *slower* than matrices without such constraints, rather than faster. The reason for this is unclear, and whether this behaviour is due to the place-and-route tools, architecture, or both is unknown.

5. Implementation

In this section the hardware performance of the generators is tested using VHDL implementations in the Stratix-II, Spartan-3 and Virtex-4 architectures.

Given a binary recurrence matrix, it is straightforward to create a hardware description that imple-

ments it. For example, the following Handel-C code segment:

```
macro expr k=6:
macro expr matrix ={{0,0,0,1,1,1},
                    {1,1,0,0,0,1},
                    {1,0,0,0,1,1},
                    {0,1,1,1,0,0},
                    {1,1,0,0,1,0},
                    {0,0,1,1,0,0}};
macro expr fb(i,row)= select (i==k,0,
  (state[i]&row[i])^fb(i−1,row)

bool state[k];
par(i=0;i<k;i++){
  state[i]=fb(0, matrix[i]);
}
```

can be used to implement the six bit generator example shown previously, or any other generator if the *matrix* and *k* constants are changed.

Figures 5, 6, and 7 give feedback matrices of a practical size in a more compact form. Each tuple within the data-set identifies the feedback taps for bit-0 through bit-*k*, with each tuple containing the zero-based offsets of the tap locations. A negative one in a tuple indicates that less than the full number of taps are used for that bit. Listing 1 gives example Handel-C code for using data-sets in this form.

```
macro proc RNG(k,numTaps,taps,oState)
{
  // make sure initial state is not zero
  static unsigned 1 state[k]={1};
  // xor of up to numTaps bits from state
  macro expr fb(i,t) =
    select (t==numTaps, 0,
    select (taps[i][t]==−1, 0
    state[taps[i][t]]^fb(i,t+1)));
  // calculate next value of bit in state
  par(i=0;i<k;i++){
    state[i]=fb(i,0);
    oState[i]=state[i];
  }
}
```

Listing 1 Example handled-C code for implementing a random number generator using the given data-sets.

```
{{6,25,29},{0,24,29},{14,30,31},{13,24,27},{13,20,28},
{10,20,-1},{8,12,26},{10,11,17},{2,18,31},{3,24,26},
{2,7,21},{14,16,27},{17,23,31},{5,9,23},{15,19,-1},
{5,8,28},{1,7,28},{0,3,20},{1,15,18},{12,16,21},
{11,21,22},{4,9,18},{11,14,22},{4,19,30},{7,12,27},
{4,17,26},{9,15,25},{2,6,-1},{3,8,29},{1,5,16},{13,22,23},
{0,25,30}}
```

Figure 5. Feedback taps for a 32-bit 3-tap generator.

For evaluation purposes 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. The area and speed measurements are implemented using one clock input pin, one reset input pin, and with all generator state bits routed to output pins. The clock rates quoted here represent flip-flop to flip-flop delay, and do not include flip-flop to pin paths. Where not enough pins are available in a package, multiple pins are multiplexed together with exclusive-ors before being routed to output pins, with the extra area excluded from the overall total. The designs are implemented using VHDL, and compiled using ISE 8.1 for Spartan-3 and Virtex-4 devices, and Quartus II 4.0 for Stratix-II devices. The built-in synthesis was used in both tool-chains, all effort levels were set to "high," and all settings for area/speed optimisation settings were set to favour area.

In all cases where the number of inputs per bit is less than or equal to the number of LUT inputs, the reported area is exactly as predicted: for 3- or 4-tap matrices each generator requires exactly *k* flip-flips and *k* LUTs. In these cases the critical path contains just one LUT, plus a routing delay that increases with matrix size, due to congestion. Figure 8 shows the changes in speed for increasing values of *k* in the three different architectures. The log-trend curve fitted through each set of points shows that the

```
{{2,4,47,48},{37,39,45,50},{9,36,44,-1},{3,16,59,63},
{18,25,34,52},{10,22,24,31},{13,32,42,46},{8,37,52,62},
{5,6,19,53},{18,33,50,60},{31,46,51,59},{7,30,36,39},
{13,21,32,35},{14,25,44,63},{13,21,47,52},{3,7,11,42},
{10,53,54,56},{2,22,49,62},{1,7,37,41},{31,33,38,60},
{20,22,29,49},{0,40,54,61},{20,38,45,51},{7,25,37,40},
{9,16,33,57},{2,3,5,56},{6,43,55,58},{1,12,49,55},
{14,21,36,58},{43,44,47,57},{10,14,20,48},{6,15,29,34},
{4,26,42,56},{12,14,48,57},{8,16,26,44},{5,24,27,51},
{19,53,54,59},{22,29,35,50},{8,10,28,41},{16,21,28,32},
{4,17,39,46},{0,24,40,46},{13,35,39,56},{5,43,62,63},
{1,45,49,60},{0,8,23,60},{4,38,42,61},{0,15,18,47},
{17,24,38,58},{17,18,35,62},{17,19,43,51},{41,48,52,63},
{11,12,19,55},{23,27,30,61},{6,29,34,57},{9,12,15,23},
{2,30,34,61},{27,32,54,55},{20,28,31,45},{9,26,30,40},
{11,26,27,58},{1,15,25,33},{23,28,53,59},{3,11,41,50}}
```

Figure 6. Feedback taps for a 64-bit 4-tap generator.

```
{{11,39,107},{5,75,-1},{38,57,70},{94,108,-1},{47,80,96},
{21,25,60},{28,77,117},{1,101,121},{55,95,-1},{16,102,114},
{79,94,113},{21,33,92},{13,14,104},{46,48,55},{13,38,73},
{15,42,49},{16,52,120},{8,57,91},{61,83,86},{37,120,121},
{2,82,95},{32,45,118},{46,86,119},{18,42,89},{48,126,127},
{2,69,124},{60,111,114},{10,89,123},{63,104,125},{56,108,110},
{82,91,119},{0,48,90},{6,7,117},{59,103,127},{59,92,100},
{14,44,51},{14,72,73},{19,39,78},{44,109,116},{54,67,104},
{10,61,100},{20,30,67},{16,69,118},{22,77,79},{41,61,98},
{27,62,98},{12,68,93},{8,41,101},{23,100,115},{26,40,68},
{43,62,109},{22,81,92},{10,29,84},{85,97,107},{27,97,124},
{22,116,124},{7,52,87},{15,32,94},{5,32,75},{28,51,89},
{11,56,123},{47,56,65},{43,70,126},{71,74,117},{20,29,91},
{93,96,111},{46,51,99},{2,27,37},{41,45,82},{12,30,88},
{3,25,54},{70,71,80},{84,119,127},{18,74,75},{31,81,105},
{4,37,63},{19,33,36},{29,34,45},{40,68,72},{12,38,112},
{23,43,52},{8,113,122},{44,83,-1},{65,123,125},{50,55,90},
{53,77,115},{42,80,85},{50,64,87},{17,39,96},{0,79,115},
{13,87,122},{30,62,106},{0,24,101},{15,60,88},{35,99,103},
{1,58,76},{31,40,78},{18,26,73},{17,64,102},{24,67,84},
{6,19,111},{33,36,114},{24,36,65},{49,105,112},{35,76,118},
{23,66,116},{26,81,106},{9,34,57},{54,122,126},{4,85,112},
{71,98,99},{6,93,105},{11,72,107},{9,53,113},{53,103,125},
{28,74,121},{58,120,-1},{7,21,106},{47,76,88},{3,31,66},
{20,59,86},{49,90,110},{34,35,97},{4,25,109},{9,63,66},
{50,78,102},{1,3,69},{17,64,110}}
```

Figure 7. Feedback taps for a 128-bit 3-tap generator.

decrease in speed is approximately logarithmic in the number of state bits. No statistically significant difference in timing is detected between 3-tap generators that supported sequential loading and unloading of state compared with those that do.

Figure 9 shows the change in area and speed as the number of taps is increased in a 64-bit generator. Once the number of inputs per exclusive-or calculation exceeds that of a single LUT, the synthesis and place-and-route tools have to start making more complex decisions about how to compute partial products. This is most striking in the 4-LUT based Virtex-4 and Spartan-3 architectures, where a 5-tap generator requires twice the LUTs of a 4-tap generator. However, the ALUTs of the Stratix-II

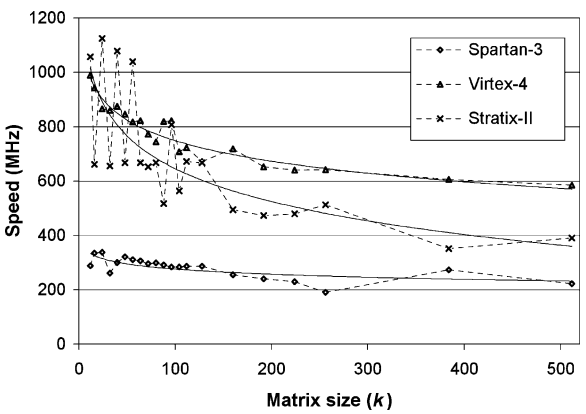

Figure 8. Clock rates (*according to critical path*) for 4-tap generators of increasing matrix size, with fitted log-trend for each family.

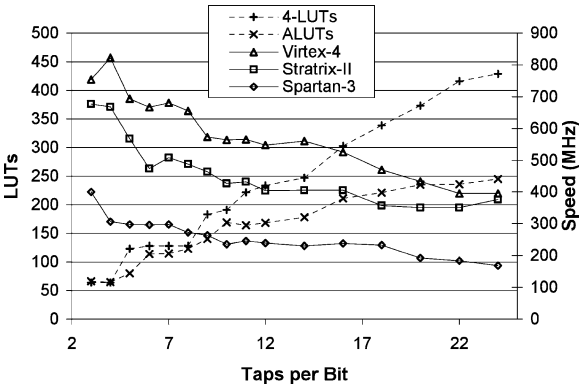

Figure 9. Changing LUT count and clock rate for 64-bit generators with an increasing number of taps.

can support more inputs per LUT, and allow more flexibility when partitioning the ALUTs to create partial products, so the number of LUTs for a given tap count is lower.

As well as requiring more area, increasing the tap counts also increases the critical path, due both to the increase in logic depth needed to implement the wider exclusive-or functions, and because the fan-out per-bit increases with the tap count. Given the increase in LUTs and decrease in clock rate, there are very few occasions where it is worth using more taps than can be supported by a single LUT. Each extra LUT that is used to support a wider exclusive-or could equally be used as a LUT plus register to increase the matrix size (rather than allowing more taps), and the subsequent increase in period is likely to provide better quality than increasing the number of taps. For example, compare the quality of the 3-tap, k=128 and 5-tap, k=64 generators in Table 3 in Section 7.

6. *Further Optimisations*

As shown in Section 7, the statistical quality of the generators shown so far is good, but suffers from the same problem as any generator based on a linear recurrence: 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 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 with support for serial loading. Because the periods of the two generators are coprime the full period 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 1 Gb sample treated as 1,000 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. Empirical Statistical Quality

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 is 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 10,000 chance that the wrong decision is made. The tests are performed by executing the matrix generators in hardware using an RC2000 (Alpha-Data ADMX-RC2) system [2] (containing an XC2V6000 FPGA), with a software wrapper to return the generated samples back to the test suites. The generators 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.

Results for the Diehard and Crush batteries are shown in Table 2, indicating the number of tests failed, and an abbreviation of each of the failed test types. The abbreviations are expanded underneath the table; for a full explanation of each test, see the Diehard [16] and Crush [13] documentation.

The 4-tap generators represent the case where the generator state does not need to be loaded (e.g. a free running generator or test vector generator), while the 3-tap generators are for use where the state needs to be loaded (e.g. for a simulation application). The third group contains results for two Combined Tausworthe generators [12], and two parallel LFSRs generated using Xilinx CoreGen.

A feature of the matrix generators is that all k bits are usable, so another test of the quality of all k/w streams of the selected generators was also performed. It was found that the streams are all of roughly the same quality, and in only one exceptional case (where $k = 256$) was the quality of one stream significantly worse than another from the same generator. In that case the stream is supplied from a set of bits with very low connectivity to the rest of the matrix, forming an almost independent stream. However, this was the only example of this type found, and a notable feature of this matrix was that it had very poor equidistribution (see Section 8).

Table 3 provides a summary of the area, speed and empirical quality of multiple generators. The first

Table 2. Failed tests for the diehard and crush test batteries for different random number generators.

Generator	k	Failed tests	
		Diehard	Crush
4-taps	32	3 (BR,DNA,OPSO)	14 (6×MR,2×LC,2×RW,CP,2×BS,MO)
	64	2 (BR,DNA)	12 (6×MR,2×LC,2×RW,CP,BS)
	96	0	10 (6×MR,2×LC,RW)
	128	0	7 (5×MR,2×LC)
	256	0	6 (4×MR,2×LC)
	512	0	4 (2×MR,2×LC)
	1024	0	4 (2×MR,2×LC)
	1248	0	2 (2×LC)
3-taps	32	5 (BR,DNA,OPSO,BS,OPERM5)	17 (6×MR,2×LC,3×RW,CP,3×BS,MO,MBO)
	64	2 (BS,OPERM5)	13 (6×MR,2×LC,2×RW,CP,2×BS)
	96	1 (OPSO)	11 (6×MR,2×LC,2×RW,BS)
	128	0	8 (5×MR,2×LC,RW)
	256	0	6 (4×MR,2×LC)
	512	0	4 (2×MR,2×LC)
	1024	0	4 (2×MR,2×LC)
	1248	0	2 (2×LC)
Lfsr	64	3 (BS,OPSO,DNA)	15 (6×MR,2×LC,3×RW,HI,CP,COL,LHR)
Lfsr	128	2 (BS,OPSO)	14 (6×MR,2×LC,COL,MB,CP,LHR,HI,AC)
Taus	88	1 (OPSO)	9 (6×MR,2×LC,CP)
Taus	113	0	6 (4×MR,2×LC)

AC Auto-correlation, *BR* binary-rank, *BS* birthday-spacings, *CP* close-pair
HI Hamming-independence, *LC* linear-complexity, *LHR* longest-head-run
MBO Multinomial-bits-over, *MO* multinomial-over, *MR* matrix-rank, *RW* random-walk

group of results shows a selection of 4-tap 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 software generators running on a 3.2 GHz P4, including the widely used Mersenne Twister (mt19937) [19]. The LUT and Flip-Flop counts only apply to the Virtex-4 implementation, although for the first two groups (4-tap and 3-tap generators) the size of the generators was identical across all three devices.

In many cases the critical path of a generator is extremely fast, and speeds of up to 1 GHz are seen in Fig. 8. However, in practise the working speed will be limited to that of the clock distribution lines, so Table 3 limits the reported generator frequency to the minimum of the critical path and the global clock net. Where clock distribution is the limiting factor the corresponding entry is italicised. Where the manufacturers do not list a maximum global clock net frequency, the maximum output frequency from the DCMs/DLLs is used.

The Diehard results reveal the slight loss in randomness in the 3-tap generators, as the 4-tap generators pass with $k = 96$, while the 3-tap generators only pass at $k = 128$. The Crush results show this as well, with the 4-tap generators passing more tests for the same k value. The parallel LFSR-160 generator gives similar quality to the 3- and 4-tap generators with $k = 64$, but requires 7 times as

Table 3. Summary of the quality, area and speed of a selection of hardware generators.

		Test failures						Virtex-4		
Generator	Period (log_2)	Diehard	Crush	FFs	LUTs	Sprtn−3 MHz	Strtx-II MHz	MHz	Gb/s	Gb/s/LUT
4-tap, k=32	32	3	14	32	32	261	422	500	16	0.50
4-tap, k=64	64	2	12	64	64	306	422	500	32	0.50
4-tap, k=96	96	0	10	96	96	283	422	500	48	0.50
4-tap, k=128	128	0	7	128	128	287	422	500	64	0.50
4-tap, k=256	256	0	6	256	256	191	422	500	128	0.50
4-tap, k=512	512	0	4	512	512	222	391	500	256	0.50
4-tap, k=1248	1,248	0	2	1,248	1,248	176	340	410	511	0.41
3-tap, k=32	32	5	17	32	32	334	422	500	16	0.50
3-tap, k=64	64	2	13	64	64	334	422	500	32	0.50
3-tap, k=96	96	1	11	96	96	314	442	500	48	0.50
3-tap, k=128	128	0	8	128	128	334	422	500	64	0.50
3-tap, k=256	256	0	6	256	256	303	422	500	128	0.50
3-tap, k=512	512	0	4	512	512	262	398	500	256	0.50
3-tap, k=1248	1,248	0	2	1,248	1,248	247	342	432	539	0.43
5-tap, k=64	64	2	12	64	123	297	422	500	32	0.26
6-tap, k=64	64	1	12	64	128	297	422	500	32	0.25
7-tap, k=64	64	1	12	64	128	298	422	500	32	0.25
8-tap, k=64	64	1	11	64	128	272	422	500	32	0.25
12-tap, k=64	64	1	11	64	234	239	403	500	32	0.14
16-tap, k=64	64	1	10	64	303	238	405	500	32	0.11
Combo,1-strm	160	0	0	307	202	181	315	380	12	0.06
Combo,2-strm	160	0	0	387	242	176	311	360	23	0.10
Taus88	88	1	9	132	129	257	413	470	15	0.11
Taus113	113	0	6	164	161	258	415	482	15	0.10
LFSR-64	64	3	15	291	321	212	422	355	11	0.04
LFSR-160	160	2	14	451	481	209	422	320	10	0.02
Generator	Period	Diehard	Crush					Pentium-4 3.2 GHz		
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	

The top three sections give results for linear recurrence generators of varying sizes and tap counts, the next section for the combined generators suggested in Section 6, and the final two sections give results for existing random number generation methods in hardware and software.

many LUTs, even with the SRL16 optimisations performed by CoreGen.

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 unexpected, as the generators in [12] are selected to have Maximal Equidistribution (i.e. a sum of dimension gaps of zero), but also have the further property of being Collision Free, so have slightly better equidistribution than the matrices used in the table. For larger periods the matrix generators achieve equal or better quality, while requiring less logic per sample generated: the *4-tap*, $k = 256$ generator is of about the same quality as Taus113, but has over eight times the pure sample rate, and achieves 5 times the sample rate per LUT used.

When high quality random number generation is considered, the LFSR based generators cannot compete due to large area and poor quality. For instance, the *combo, 2-strm* 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.

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 1,200. For evaluation purposes a period just over 1,200 is chosen, just to check that it could be passed. A better solution is the modifications suggested in Section 6, using in the combo generators.

8. Theoretical Statistical Quality

The equidistribution test provides a theoretical quality metric that applies to a generator's entire output sequence, as opposed to empirical tests that can usually only test a very small sub-sequence. The test determines how evenly successive t-tuples of random outputs fill a t-dimensional hyper-cube, by partitioning the hypercube into multiple buckets and counting the number of times each bucket is hit [21]. By using properties of linear recurrences it is possible to calculate the equidistribution of a generator over it's entire sequence, without having to manually generate and classify each output.

Specific measures of quality are made by splitting the t-dimensional hyper-cube into 2^l equal sized segments, where $l \leq k$ (k is the number of binary bits in the generator state). This means that the 2^k possible t-tuples are assigned to a total of 2^{tl} buckets in the t-dimensional hyper-cube. A generator is said to be (t, l)-equidistributed if each bucket in the hypercube contains 2^{k-tl} points. For a given resolution of l, let t_l be largest dimension t for which a generator is (t, l)-equidistributed, with an upper bound on t_l of $t_l^* = \lfloor k/l \rfloor$. The quality of a generator can then be measured by using the dimension gap $\delta_l = t_l^* - t_l$. For a given resolution l a low value of δ_l indicates a good equidistribution, with $\delta_l = 0$ indicating the best possible equidistribution.

Each resolution l measures the quality of the l most significant bits of a generated sequence, so $\delta_2 = 0$ would indicate that the two most significant bits of the sequence have the optimum distribution. A measure of quality across all bits in the output sequence is provided by the worst-case dimension gap Δ_∞ and the sum of dimension gaps Δ_1:

$$\Delta_\infty = \max_{1 \leq l \leq w} \delta_l \Delta_1 = \sum_{l=1}^{l=w} \delta_l \tag{3}$$

Together these two measures characterise the optimality of a generator across all output bits, with $\Delta_1 = 0$ indicating a maximally equidistributed generator.

Calculating δ_l over the entire output sequence is not possible for all types of generator (such as Cellular Automata), and in those cases only an empirical measure of local sub-sequence equidistribution can be calculated. In the case of binary linear recurrences, it is possible to calculate δ_l using properties of the state-transition matrix. From a given state s_i, it is possible to directly calculate s_{i+j}, or any individual bit within it, using the matrix A^j. This allows the tl bits that form each t-tuple with resolution l to be expressed as a $tl \times k$ matrix $E_{t,l}$. It can be shown that a necessary and sufficient condition for (t, l)-equidistribution is that $E_{t,l}$ have full rank [8, 12]. In this way it is possible to directly calculate Δ_∞ and Δ_1 for binary linear recurrences.

In the case of the hardware binary linear recurrences discussed in this paper w is typically less than k, but the distribution of the entire k-bit state is still of interest. Table 4 summarises the equidistribution of the best matrices found for different values of k. Δ_∞ and Δ_1 are shown both for a likely output width of $w = 32$, and for $w = k$. Also included are the weights of the characteristic polynomials. In all cases the weight is close to $k/2$, an indicator of good statistical quality.

Because the search process used to find matrices is stochastic, there is no guarantee that just because no maximally equi-distributed generators are quoted in Table 4 that they do not exist. Figure

Table 4. Equidistribution of best 3- and 4-tap generators found through random search.

Generator		$w = 32$		$w = k$		Weight of $P(z)$
k	Taps	Δ_∞	Δ_1	Δ_∞	Δ_1	
32	3	1	1	1	1	16
32	4	1	1	1	1	14
64	3	1	6	1	6	30
64	4	1	2	1	2	34
96	3	1	4	1	15	44
96	4	1	3	1	12	42
128	3	1	2	1	18	62
128	4	1	4	1	7	68
256	3	2	7	2	27	124
256	4	2	6	2	22	122
64	3	1	6	1	6	30
64	4	1	2	1	2	34
64	5	0	0	0	0	28
64	6	0	0	0	0	28
64	7	1	1	1	1	36
64	8	1	1	1	1	32
64	12	1	1	1	1	28
64	16	1	1	1	1	26

10 shows the cumulative distribution of Δ_1 for a 32 bit output sequence. It is clearly much easier to find a well equidistributed matrix with large k than it is for small k. Figure 11 shows the cumulative distribution of Δ_1 for 64-bit generators using different numbers

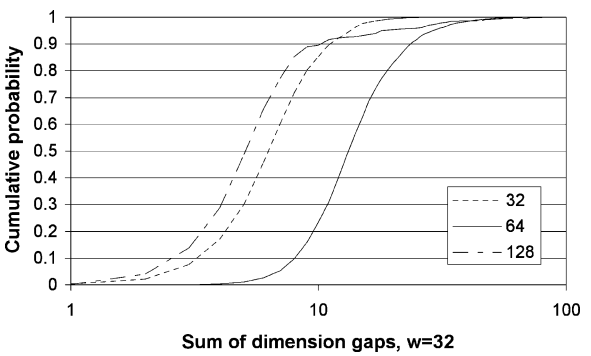

Figure 10. Cumulative probability distribution of Δ_1 for k={32,64,128} and w=32.

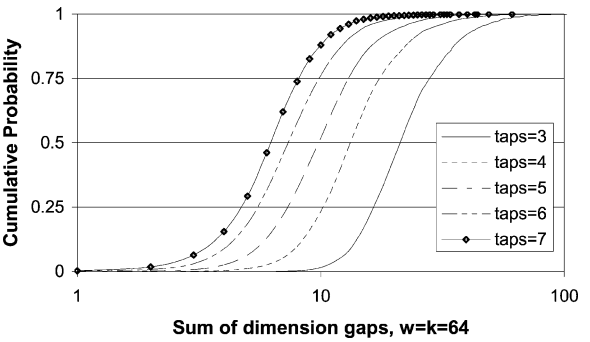

Figure 11. Cumulative probability distribution of Δ_1 for generators with differing numbers of taps and $k = w = 64$.

of taps. As the number of taps is increased it becomes much more likely that a generator with good equidistribution is found. However, a generator with more taps requires a recurrence matrix with a larger number of ones, increasing the time taken to generate each matrix. This results in a sweet-spot of around 5 or 6 taps, where the probability of finding a maximally equidistributed generator balances the time taken to find each full-period generator, explaining why these are the only two such generators in Table 4.

9. 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

Table 5. Comparison of two 4-tap generators, the additive combination generator, a combined Tausworthe generator, and the software Mersenne Twister.

Generator	Period	Quality	Gb/s	FF/LUT
4-tap,k=128	128	Medium	64	128/128
4-tap,k=512	512	Good	256	512/512
Combined	160	Excellent	23	387/242
Taus113	113	Good	15	164/161
Mt19937	19,937	Excellent	2	N/A

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. The theoretical properties of these matrices, as measured through equidistribution, are very good, and maximally equidistributed generators within this family of generators can be found.

Table 5 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 360 MHz in the Virtex-4 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. Improving the efficiency of the search process should increase the speed at which full-period matrices can be found, making it possible to find more maximally equidistributed and collision free generators. This could be achieved by using canonical labels for matrices in order to detect matrices that have already been tried, and to allow for exhaustive searches for state-transition matrices.

Different FPGA families offer opportunities for increasing quality or reducing area using architecture specific components. For instance, the Virtex SRL16 could be used to provide high periods when not all bits of the state will be consumed, while the Stratix-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. P. Alfke, “Efficient Shift Registers, LFSR Counters, and Long Pseudo-random Sequence Generators.” Technical Report, Xilinx, Inc., 1996.
2. Alpha Data, *ADM-XRC SDK User Guide 4.3.1*, 2003.
3. Altera Corporation, *Stratix II Device Handbook, Volume 1*, 2005.
4. R. Andraka and R. Phelps, “An FPGA Based Processor Yields a Real Time High Fidelity Radar Environment Simulator,” in *Conference on Military and Aerospace Applications of Programmable Devices and Technologies*, 1998.
5. J. Chen, J. Moon, and K. Bazargan, “Reconfigurable Read-back-signal Generator Based on a Field-programmable Gate Array,” *IEEE Transactions on Magnetics*, vol. 40, no. 3, 2004, pp. 1744–1750.
6. S. Duplichan, PPSearch: A Primitive Polynomial Search Program, http://users2.ev1.net/~sduplichan/primitivepolynomials/, 2003.
7. V. Fischer and M. 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*, Springer, Berlin Heidelberg New York, 2003, pp. 415–430.
8. M. Fushimi and S. Tezuka, “The K-distribution of Generalized Feedback Shift Register Pseudorandom Numbers,” *Communications of the ACM*, vol. 26, no. 7, 1983, pp. 516–523.
9. M. George and P. Alfke, Linear Feedback Shift Registers in Virtex Devices, Technical Report, Xilinx, Inc., 2001.
10. P. D. Hortensius, R. D. McLeod, and H. C. Card, “Parallel Random Number Generation for VLSI Systems Using Cellular Automata,” *IEEE Transactions on Computers*, vol. 38, no. 10, 1989, 1466–1473.
11. D. E. Knuth, *Semi-numerical Algorithms, Volume 2 of the Art of Computer Programming*, 2nd edition, Addison-Wesley, Reading, MA, 1981.
12. P. L’Ecuyer, “Maximally Equidistributed Combined Tausworthe Generators,” *Mathematics and Computation*, vol. 65, no. 213, 1996, pp. 203–213.
13. P. L’Ecuyer and R. Simard, TestU01 Random Number Test Suite, http://www.iro.umontreal.ca/~simardr/indexe.html.
14. D. Lee, J. Villasenor, W. Luk, and P. Leong, “A Hardware Gaussian Noise Generator Using the Box-muller Method and Its Error Analysis,” To Appear in *IEEE Transactions on Computers*, 2006.
15. D.-U. Lee, W. Luk, J. D. Villasenor, and P. Y. Cheung, “A Gaussian Noise Generator for Hardware-based Simulations,” *IEEE Transactions on Computers*, vol. 53, no. 12, 2004, pp. 1523–1534.(December)
16. G. Marsaglia, The Diehard Random Number Test Suite, http://stat.fsu.edu/pub/diehard/, 1997.
17. G. A. Marsaglia and L. Tsay, “Matrices and the Structure of Random Number Sequences,” *Linear Algebra and its Applications*, vol. 67, 1985, pp. 147–156.
18. M. Matsumoto and Y. Kurita, “Twisted GFSR Generators II,” *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 3, 1994, pp. 254–266.

19. 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, 1998, pp. 3–30.(January)
20. A. Negoi and J. Zimmermann, "Monte Carlo Hardware Simulator for Electron Dynamics in Semiconductors," in *International Annual Semiconductor Conference*, Sinaia, Romania, 1996, pp. 557–560.
21. F. Panneton and P. L'Ecuyer, "On the Xorshift Random Number Generators," To appear in *ACM Transactions on Modeling and Simulation*, 2005.
22. F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved Long-period Generators Based on Linear Recurrences Modulo 2," To appear in *ACM Transactions on Mathematical Software*, 2005.
23. B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider, "FPGA Implementation of Neighborhood-of-four Cellular Automata Random Number Generators," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, New York, 2002, pp. 106–112.
24. V. Shoup, Ntl: A Library for Doing Number Theory, http://www.shoup.net/ntl/.
25. R. C. Tausworthe, "Random Numbers Generated by Linear Recurrence Modulo Two," *Mathematics and Computation*, vol. 19, no. 90, 1965, pp. 201–209.
26. K. H. Tsoi, K. H. Leung, and P. H. W. Leong, "Compact FPGA-based True and Pseudo Random Number Generators," in *IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE Computer Society, Washington, DC, 2003, p. 51.
27. S. Wolfram, "Random Sequence Generation by Cellular Automata," *Advances in Applied Mathematics*, vol. 7, no. 2, 1986, pp. 123–169.
28. Xilinx, Inc., *Virtex-II Platform FPGAs: Complete Data Sheet*, 2000.
29. 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 *International Conference on Field Programmable Logic and Applications*, IEEE Computer Society , 2005, pp. 275–280.
30. G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk, "Reconfigurable Acceleration for Monte Carlo Based Financial Simulation," in *International Conference on Field-Programmable Technology*, IEEE Computer Society , 2005, pp. 215–224.

David B. Thomas received the MEng and Ph.D. degrees in computer science from Imperial College, in 2001 and 2006, respectively. He likes Imperial so much that he stayed on, and is now a post-doctoral researcher in the Custom Computing group. Research interests include FPGA-based Monte-Carlo simulations, algorithms and architectures for uniform and non-uniform random number generation, and financial computing.

Wayne Luk received the MA, MSc, and DPhil degrees in engineering and computer science from the University of Oxford, Oxford, United Kingdom. He is a professor of computer engineering, Department of Computing, Imperial College London and leads the Custom Computing Group there. His research interests include theory and practice of customizing hardware and software for specific application domains, such as graphics and image processing, multimedia, and communications. Much of his current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays. He is a member of the IEEE.