Compact FPGA-based True and Pseudo Random Number Generators

K.H. Tsoi, K.H. Leung and P.H.W. Leong
{khtsoi,khleung,phwl}@cse.cuhk.edu.hk
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, NT Hong Kong

Abstract

Two FPGA based implementations of random number generators intended for embedded cryptographic applications are presented. The first is a true random number generator (TRNG) which employs oscillator phase noise, and the second is a bit serial implementation of a Blum Blum Shub (BBS) pseudorandom number generator (PRNG). Both designs are extremely compact and can be implemented on any FPGA or PLD device. They were designed specifically for use as FPGA based cryptographic hardware cores. The TRNG and PRNG were tested using the NIST and Diehard random number test suites.

1 Introduction

The random number generator (RNG) is an important cryptographic primitive widely used for one time pads [24], key generation (e.g. [21]) and authentication protocols (e.g. [28]). The security of such systems rely on the assumption that future values in the random number sequence cannot be predicted from the observed sequence. There are two types of random number generators commonly used for cryptographic applications. The true random number generator (TRNG) derives its output from a physical noise source whereas a pseudorandom number generator (PRNG) expands a relatively short key (possibly from a TRNG) into a long sequence of seemingly random bits based on a deterministic algorithm. A cryptographically secure random bit generator (CSRBG) is one which produces sequences for which there is no polynomial time algorithm which, on input of the first l bits of the output sequence s, can predict the $(l+1)st$ bit of s with a probability significantly greater than $\frac{1}{2}$ [16].

Field programmable gate array (FPGA) devices have been successfully used for the implementation of cryptographic hardware, some examples being the data encryption standard (DES) [18], advanced encryption standard (AES) candidate finalists [3], IDEA [17] and RSA cryptography [26]. In these and other implementations, FPGAs had advantages in performance, design time, power consumption, flexibility, cost or area over comparable microprocessor and very large scale integration (VLSI) based systems.

In this work, FPGA based implementations of a TRNG and a PRNG are presented. These designs are intended for integration with other FPGA based cryptographic hardware to produce embedded cryptosystems on a single FPGA. Apart from achieving a higher level of integration, keeping the critical random number generation operations internal to the device achieves better security since these data do not need to be passed to the FPGA via the pins.

In many applications, highly secure random numbers are required only at very low bit rates, perhaps to generate a single key for the lifetime of the application. An example is public key cryptography where, once a key pair is generated, the same key is used for subsequent applications. The TRNG and PRNG reported in this paper are designed for low bit rate applications and both are able to generate highly secure random numbers while occupying minimal resources. They are particularly suitable for applications where integration of the RNG and other cryptographic algorithms on the same FPGA is required.

Previous implementations of TRNGs will be reviewed in Section 2. For the TRNG, oscillator phase noise was used. This mechanism was chosen since the RNG can be constructed from mostly digital components and is thus suitable for FPGA implementation. Unlike other implementations using this approach [22], our implementation uses a very high frequency clock (up to 400 MHz) and does not require a scrambler to achieve good random output. The only other single chip FPGA based TRNG of which we are aware uses analogue phase locked loop (PLL) jitter and can only be implemented on an FPGA with this feature [5]. In contrast, the method presented in this paper can be implemented on any FPGA device.

The PRNG presented is a novel bit serial implementation of the Blum, Blum and Shub (BBS) [12] PRNG. This

algorithm was chosen for its extremely compact circuit, yet BBS is a CSRBG if the assertion that that there is no polynomial time algorithm for integer factorization remains true (an unproven assumption). BBS is believed to be one of the strongest CSRBG algorithms.

The rest of the paper is organized as follows, traditional techniques used for random number generation (RNG) are reviewed in Section 2. The basic algorithms used for the TRNG and PRNG described in this paper are then introduced in Section 3. The architecture of the RNGs and design details are presented in Section 4. The performance of the design and the quality of the resulting output is reported and evaluated in Section 5. Finally, conclusions are drawn in Section 6.

2 Review of Random Number Generation Techniques

2.1 True Random Number Generators

Given the importance of random number generation, surprisingly few hardware implementations of TRNGs have been reported. There are three commonly used techniques in the literature, namely oscillator sampling, direct amplification and discrete time chaos. In the oscillator sampling approach, period variation (i.e. oscillator jitter) in a low frequency clock of low quality factor (Q) is exploited by using it to sample a high frequency clock. The direct amplification technique digitizes thermal or shot noise, using a amplifier and comparator. Finally, chaotic systems can be used to produce TRNGs.

In 1984, Fairfield, Mortenson and Coulthart [22] developed the first integrated RNG based on oscillator phase noise. In the design, a high frequency oscillator was sampled using a low frequency oscillator. After removing duty cycle biases via a parity filter, the flip flop output was fed into a linear feedback shift register (LFSR) based scrambler. The design generated 27 bps using a 1000 Hz low frequency clock. Although our random source uses the same mechanism, our random number design does not require the digital scrambler used in their design.

The Intel RNG is part of the Intel 8xx chipset starting with the Intel 810 and is implemented in the Intel 82802 Firmware Hub Device (FWH). It uses amplified thermal noise to drive a voltage controlled oscillator (VCO), and oscillator sampling is used to detect the phase noise of the VCO to produce a digital random source [10].

To the best of the authors' knowledge, the only previously reported FPGA implementation of a TRNG was an implementation by Fischer and Drutarovsky [5] using a variation of oscillator sampling. Their design was based on the randomness of jitter in an analogue phase locked loop (PLL) and a decimator was used to ensure that at least one sample affecting jitter was included in every output data. The design was implemented on an Altera APEX EP20K200-2X FPGA with a 33.3 MHz external clock. With an 88.245 MHz internal clock, it can generate 69 kbps. For FPGAs such as the Altera APEX E and APEX II devices which have internal PLLs, this approach requires no external components. The disadvantage of this approach is that few FPGAs have this feature.

True random number generators based on chaotic systems can lead to very compact CMOS implementations. In 2001, Stojanovski *et al.* [27] implemented an analog chaos-based RNG in a 0.8 μm CMOS process utilizing switched current techniques. The estimated output bit rate of this design was 1 Mbps. Andrea Gerosa *et al.* [7] also implemented a RNG based on a chaotic system. Their design with a pipelined ADC (analog-to-digital converter) occupied 2.2 mm^2 silicon area and the design can generate 8-bits of data using a 20 MHz clock.

Petrie *et al.*, combined oscillator sampling, direct amplification and discrete time chaos to produce an analog VLSI chip which was robust to power supply noise and substrate signal coupling [19]. Implemented in 2 μm CMOS, the chip could produce random numbers at 1.4 Mbps. The design occupied an area of 1.5 mm^2 and dissipated 3.9 mW of power.

2.2 Pseudorandom Number Generators

There are many methods to generate pseudorandom sequences, and the classical software based methods, all of which can be implemented in hardware, are described in Knuth [11]. In this section, hardware implementations of PRNGs will be reviewed.

A common method of producing a PRNG is to use the output of a linear feedback shift register (LFSR) [11]. Although this technique has good statistical properties and leads to very efficient hardware implementations, the Berlekamp–Massey algorithm can be used to efficiently deduce the connection polynomial from the LFSR's output sequence, making it unsuitable for cryptographic applications [16].

In 1986, Wolfram [31] proposed a method to generate random numbers by connected cellular automatas (CA). Hortensius *et al.* [8] proposed a VLSI implementation of a parallel 1-D cellular automata. The 30-bit hybrid CA design was about 2.1 times larger than a 30-bit LFSR (linear feedback shift register) RNG while offering better randomness and faster clock rate due to nearest neighbor wiring.

In 2002, P. Martin [15] evaluated different PRNGs implementation on FPGAs. He showed that using multiple LFSRs with different initial values can give more random results that a single LFSR.

FPGA based cryptographic PRNGs hardware have also

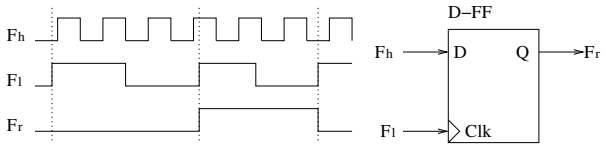

Figure 1. Oscillator sampling using D-type flip-flop.

been proposed. Barry Shackleord *et al.* presented RNGs based on neighborhood-of-four cellular automata [25]. The design made use of the 4-input lookup tables (LUTs) in Xilinx FPGA to fully utilize the hardware and can generate 64-bit random numbers at a frequency as high as 230 MHz. Another FPGA implementation of PRNG was introduced by Robert K. Watkins *et al.* in 2001 [30]. Their design used a Genetic Algorithm (GA) to generate a set of PRNGs and FIPS-140 was used as a fitness function in the evolution. This design, implemented on an XESS XSV800 Virtex prototyping board, relied on run time reconfiguration. The final product of the evolution is a PRNG.

It is not possible to prove a sequence is random. Some basic tests were introduced by Knuth [11]. A compact and preliminary test suite was defined in FIPS-140 by the National Institute of Standards and Technology (NIST). NIST proposed a more comprehensive random and pseudorandom number generator test suite for cryptographic applications in 2001 [1]. The Diehard test developed by Marsaglia [14] is widely considered to be one of the most stringent RNG tests.

3 RNG Algorithms

3.1 TRNG Design

The TRNG operates by sampling an accurate high frequency clock, F_h, with an unstable low frequency clock, F_l. This was done using an edge-triggered D-type flip-flop as shown in Figure 1 with F_l as the clock input and F_h as the data input. The output rate is at the frequency of the the slow clock, F_l.

There are several factors which affect the randomness of the output [22]. The first situation is that the duty cycle of F_h may not be 50%. In this situation, F_r will have unequal probability of being zero or one. An N-bit parity filter [4, 22] can be used to deskew a non-uniform distribution. If the ratio of ones to zeroes in the raw random bitstream is $p : q$ then the probability that the parity will be one or zero is the sum of the odd or even terms of the binomial expansion of $(p + q)^N$. This sum can be evaluated to calculate the probability of a one at the output of the parity filter and is $\frac{1}{2}((p + q)^N + (p - q)^N)$. Since $p + q = 1$, this expression reduces to $\frac{1}{2}(1 + (p - q)^N)$. As n increases, this expression tends to 0.5.

The second factor is the selection of clock frequency. If the variation of the period in F_l is not large enough, there will be correlation between bits and so the value of the output can be predicted to some extent from the previous values. Previous research has shown that the standard deviation of the period of F_l should at least be 0.75 times the period of F_h [22]. Thus increasing F_h and reducing F_l leads to more randomness. The effect of the frequency of F_l on the quality of the random numbers generated will be addressed experimentally in Section 5.

A third factor affecting the quality of the RNG is the random source itself. As there are both periodic and aperiodic electromagnetic noise inside a computer system, there may be correlation in the output sequence as the result of coupling of periodic noise from the power supply, clocks, crosstalk, thermal effects etc. This issue is not addressed in this work.

3.2 PRNG Algorithm

The following equation generates the BBS sequence X_i where i is a positive integer:

$$X_{i+1} = X_i^2 \ mod \ M \tag{1}$$

where *M* is a product of two large prime numbers *p* and *q*, which both have a remainder of 3 when divided by 4. X_0 is a seed which is co-prime with *M*. As proven in [12], a deterministic algorithm to compute the unique quadratic residue $X_{-1} \ mod \ M$ such that $(X_{-1})^2 \ mod \ M = X_0$ requires the knowledge of the prime factors of M. Thus M can be made public as long as *p* and *q* are kept secret and the difficulty of deducing the output of the PRNG is as difficult as factorizing M.

Currently, the best general purpose factoring algorithm is the Number Field Sieve (NFS) which has a runtime approximately $O(e^{1.9(\ln n)^{1/3}(\ln \ln n)^{2/3}})$ [13, 23]. To date, the largest numbers (without special properties) that have been factored were 155-digits (512-bits) in length. Thus 1024-bit numbers are considered very secure [23] and are routinely used for high security applications.

The BBS algorithm is appropriate for use in cryptographic applications since it has a strong security proof which relates the quality of the generator to the difficulty of integer factorization [12]. Although the original algorithm only produced 1-bit per iteration, Vazirani and Vazirani [29] showed that one can safely use at least $\lfloor log_2(log_2(M)) \rfloor$ bits of X_i while maintaining equivalent security. The typical size of *M* ranges from 256 to 1024-bits [1]. Using a larger *M* will increase the number of available bits in each

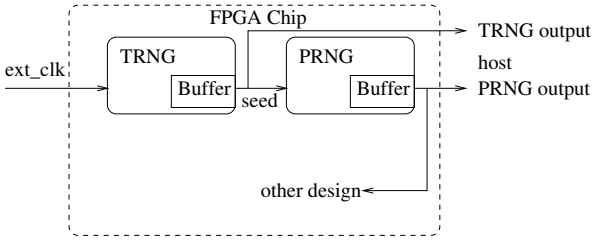

Figure 2. TRNG and PRNG as a subsystem for an FPGA-based application.

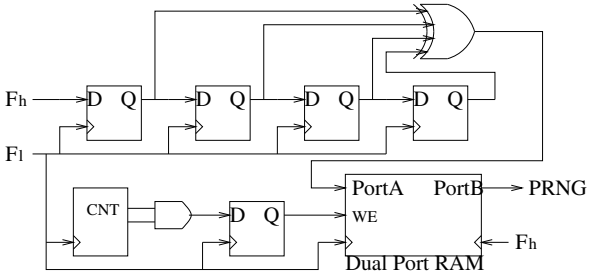

Figure 3. TRNG circuit showing digital mixing, parity filter and output buffer.

iteration at the expense of increased area and computational requirements.

4 Implementation

In this section, the implementation details of the TRNG and PRNG are presented. The complete TRNG and PRNG was designed on a single FPGA with the only off-chip components being two resistors and a capacitor for the TRNG low frequency oscillator. The TRNG and PRNG operate independently. Figure 2 shows the relation between the two parts. The TRNG first fills its buffer with random bits. This buffer can be used as a source of random bits and can also be used as a seed for the PRNG. The output of PRNG is also stored in a buffer. A host computer interface permits retrieval of the data in either of the buffers.

4.1 TRNG Circuit

The TRNG circuit of Figure 3 was used for the design. F_h is a high frequency clock and F_l is a low frequency clock generated by an external RC oscillator circuit. A parity filter with 4 stages was applied to the raw random bit stream to correct for any duty cycle biases. Note that since the parity filter must use n independent events, the output rate is reduced by a factor of n. This is implemented by only producing one write enable every n cycles as shown in the schematic.

Figure 4 shows the schematic for the free running multivibrator used to implement the F_l oscillator [6, 2]. The period of the oscillator will be affected by background noise which is the source of randomness in the design. R2 is used to limit the current when Y has an excursion beyond the supply rails i.e. without R2, the voltage at Y would be limited to between V_{dd} and GND due to the FPGA's input protection diodes. R2 should be chosen to be much larger than R1 so that there is no appreciable discharge of C through R2. Note that the inverter and two buffers in the low frequency oscillator were implemented using the input/output blocks (IOBs) of the FPGA. The IOBs have hysteresis which serve to reduce noise due to slow changing inputs of the oscillator. It is possible to add an enable to the oscillator so that it can be switched off to save power.

Figure 5 shows the output of node Y of the oscillator as measured with an oscilloscope. The highest voltage reached at node Y occurs just after node X switches from low to high. Since V_X increases by V_{dd}, V_Y increases by the same amount to $V_Y = \frac{3V_{dd}}{2}$. Charge at node Y then decays through R1 towards GND (via the output of the inverter) with an exponential decay $V_Y(t) = \frac{3V_{dd}}{2}e^{-t/(RC)}$. When V_Y reaches the switching point of the bottom buffer, i.e. $V_Y = Vdd/2$, the buffer will switch, causing V_X to drop by V_{dd}. This is turn causes V_Y to drop to $-V_{dd}/2$. The voltage then rises towards V_{dd} (since the inverter's output is at V_{dd} until the switching point of the bottom buffer is again reached. This process repeats indefinitely and is thus an oscillator.

The resulting half-period $T/2$ can be calculated as $\frac{V_{dd}}{2} = \frac{3V_{dd}}{2}e^{\frac{-T/2}{RC}}$ from which the period of the oscillator $T = RC\ln(9)$ can be derived.

The dual port BlockRAM acts as both a buffer and interface. The random bit stream is written to the memory through port A via the F_l clock. The PRNG circuit reads the output via port B using the F_h clock. The TRNG circuit also contains a counter (not shown in Figure 3) whose output is used as the address for the BlockRAM.

4.2 BBS PRNG

The implemented BBS PRNG used an M which was 1024-bits in length. The size and security of the BBS PRNG can be changed by appending more registers and increasing the counter size. The modulus *M* was hardwired in the design.

Figure 6 shows the data path of BBS PRNG. Note that all connections are one bit in width. The main computation element in the PRNG is a bit serial arithmetic logic unit

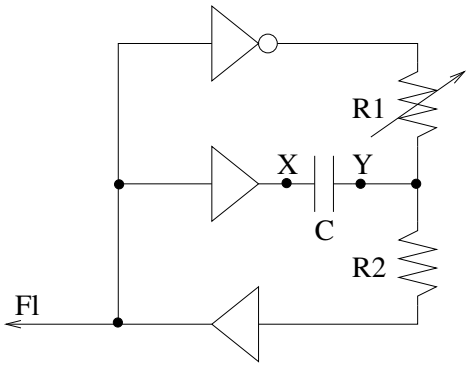

Figure 4. Low frequency clock circuit. Note that the inverter and two buffers in the oscillator circuit were implemented using the input/output blocks (IOBs) of the FPGA.

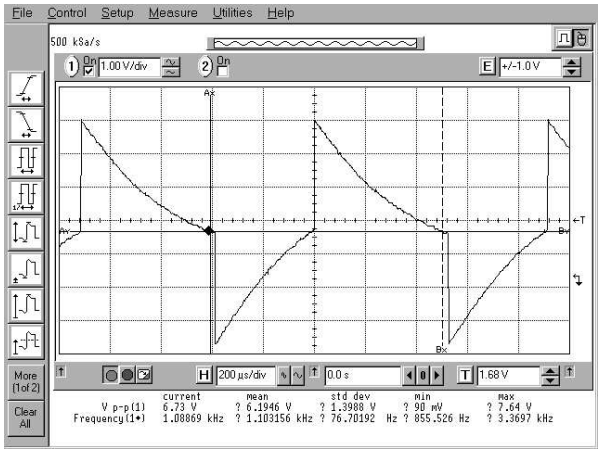

Figure 5. Oscilloscope trace of oscillator node Y.

(ALU). The signal *op* selects its operation mode, namely:

$$ALUoutput = \begin{cases} A + B + C_{in} & \text{if op = 0 and sub = 0} \\ B - A & \text{if op = 0 and sub = 1} \\ B + C_{in} & \text{otherwise} \end{cases}$$

There are 4 1024-bit shift registers in the design: M, X, Y and Z. Register M stores the value of *M* which will not be changed. Register X stores the value of X_i. This value is initialized to a random seed from the TRNG and refreshed after each iteration. Register Y and Z can be combined to form a 2048-bit register, register YZ, to store the temporary results of ALU operations. All registers are constructed by cascading SRL16E components [32] to reduce area consumption where the SRL16E element is a single LUT configured as a 16-bit shift register with enable.

Table 1. Register values for the validation operation.

Register	Value before validation	Value after validation
M	M	M
X	M	don't care
Y	X_0	1
Z	X_0	X_0

There are two internal flag registers: z_flag and 1_flag which are asserted when the 1024-bit output of the ALU is 0 or 1 respectively. These two flags are implemented in a serial fashion and are used by the control finite state machine (FSM).

The BBS PRNG performs three functions: seed validation, squaring and modulo operations. Seed validation is performed once only during initialization, and after that, a squaring and modulo operation are performed each iteration to produce X_i, the least significant $\lfloor log_2(log_2(M)) \rfloor$ bits of which are used as random data. Although one could safely use the least significant 9-bits of X_i ($M < 2^{1024}$), 8-bits were chosen to simplify interfacing.

4.2.1 Seed Validation

As described in the Section 3.2, the BBS algorithm requires that the seed, X_0 be co-prime with the modulus *M*. Euclid's method [11] was used for computing $gcd(X_0, M)$. The validation operation is repeated with different random seeds X_0 (provided by the TRNG) until a seed for which $gcd(X_0, M) = 1$ is found. The seed validation implementation is described by the following pseudocode:

```
seed_validation() {
get_seed:
    X = modulus;     // backup M
    Y = read(TRNG); // get X_0
gcd_sub:
    Y = Y - X;
    if (Y == 1) return(seed = X_0);
    if (Y == 0) goto get_seed;
    if (Y < 0) Y = Y + X;
    swap X, Y;
    goto gcd_sub;
}
```

Table 1 shows the contents of each register before and after the validation process.

All operations are performed in a serial manner. The least significant bits of registers X and Y are passed to the

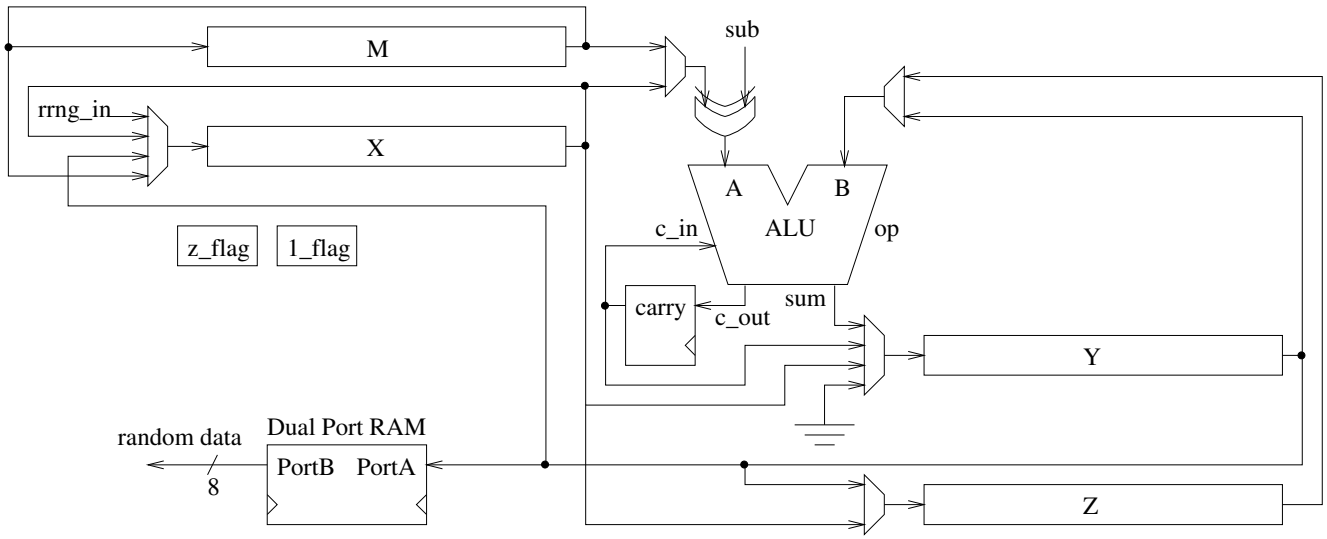

Figure 6. BBS PRNG datapath.

ALU as operands and the registers are right shifted after each clock cycle. Thus a single +/- operation takes 1024 clock cycles. The ALU's carry flag is used to test if $y - x < 0$.

4.2.2 Squaring

Table 2 shows the values of the registers before and after the squaring operation and the following pseudocode describes the procedure for performing a squaring. Note that 1024 cycles are required to perform the *add* on the line labeled L1.

```
square(X, YZ) {
    repeat 1024 times {
        if LSB(Z) = 1 then
L1:         Y = Y + X;
        else
            Y = Y;
        shift_right_one_bit(YZ);
    }
}
```

4.2.3 Mod Operation

After squaring, YZ contains X_i^2. This is reduced modulo M by restoring division until a negative result is generated, producing $X_i^2 \; mod \; M$. There are faster methods for finding the remainder but this method was chosen since it occupies a very small circuit area. The following pseudocode performs the *mod* operation.

Table 2. Register values for the squaring operation.

Register	Value before Mul	Value after Mul
M	M	M
X	X_i	X_i
Y	0	$(X_i^2)_{[511:256]}$
Z	X_i	$(X_i^2)_{[255:0]}$

```
mod(M, YZ) {
    repeat 1024 times {
        Y = Y - M;
        if (Y < 0) then
            Y = Y + M;
L1:     YZ = shift_left_1bit(YZ);
    }
}
```

Table 3 shows the values of registers before and after the *mod* process.

One implementation detail should be noted. In the pseudocode, register YZ is shifted left by one bit. In the actual hardware this is not implemented directly since the shift register employs SRL16E components which can only be used to shift in one direction. In most other operations including the validation, squaring and backup, the registers shift in the right direction, the line labeled L1 being the only exception in the design. In order to make the hardware simple and uniform, the shift left operation is transformed to a rotate right of 2047-bits. It could also have been imple-

Table 3. Register values for the mod operation.

Register	Value before Mod	Value after Mod
M	M	M
X	X_i	X_i
Y	$(X_i^2)_{[511:256]}$	$X_i^2 \ mod \ M$
Z	$(X_i^2)_{[255:0]}$	don't care

Table 4. Timing analysis of PRNG.

Operation	Cycles
1. X = x*x	n^2
2: X % M (n iterations)	
2.1. n-bit sub	n
2.2. n-bit restore	$0.5n$ (average)
2.3. shift left = rot $2n$ bits	2n
TOTAL	$n^2 + n(3.5n) = 4.5n^2$

mented by using D-type flip-flops for the Y and Z registers, however, since 16 flip-flops are required to implement a single SRL16E, the size of register Y and Z will grow by a factor of 16 so this approach was not adopted.

4.2.4 Output and Restore

After the *mod* operation, the result, $X_i^2 \ mod \ M$, is stored in register Y. The PRNG output is produced from 8 least significant bits of register Y which are shifted to the output buffer, forming the pseudorandom bit stream for the current iteration.

To restore the registers' values for the next iteration, Y is then copied to X and Z. At the same time, zero is shifted into Y and the flag and carry registers are cleared. After restoring, the values in the registers are as shown in the second column of Table 2.

4.2.5 PRNG Timing

Table 4 gives a breakdown of the number of cycles required for each step of the PRNG generation process. In total, each iteration of the PRNG requires $4.5n^2 + n$ clock cycles, where n is the size of the modules in bits.

5 Results

An implementation of the RNG was synthesized and implemented using Synopsys FPGA Compiler and Xilinx Alliance respectively. The FPGA platform used was a Pilchard FPGA card [20] which uses the SDRAM bus instead of the PCI bus used in conventional FPGA boards, and the external passive components used for the low frequency oscillator (R1, R2 and C in Figure 4) were built on a daughter card. The FPGA used was a Xilinx Virtex XCV300E-8 device.

Table 5. Implementation summary (Xilinx XCV300E-8).

Design Design	Period (ns)	Slices (% XCV300)	BRAM (% XCV300)
TRNG	2.310	15 (1%)	1 (3%)
PRNG	7.212	307 (10%)	1 (3%)
Both	7.280	310 (10%)	2 (7%)

Table 5 summarizes the resource utilization and performance of the TRNG and 1024-bit PRNG design including the host interface. The design occupies less than 10% of the logic resources of a Xilinx Virtex XCV300E-8 FPGA. The high frequency clock of the TRNG can operate at over 400 MHz, and the PRNG operates at 133 MHz. In the experiments described below, the PRNG was clocked at 133 MHz and the TRNG at 266 MHz by doubling the 133 MHz clock using a doubled delay locked loop (DLL). The frequency of the low frequency clock is variable.

For n=1024, each iteration requires 35 ms. Since the last 8-bits are used as random data, the throughput of the design is 225 bps.

5.1 TRNG Low Frequency Clock

For all experiments in this and the following randomness tests, a $50\ V$ monolithic ceramic $0.01\mu F \pm 20\%$ bypass capacitor was used for C and all resistors were discrete carbon resistors, A much larger period variation was observed if a variable resistor was used for R1, so discrete valued resistors were used for all measurements. The values of R1 ranged from 90 Ω to 2 $k\Omega$ resulting in frequencies of 18-265 kHz. The R2 resistor was chosen to be 30 $k\Omega$. The square wave output of the low frequency clock was buffered in the FPGA and the buffer output was measured using a digital oscilloscope to obtain statistics on the period variation. The buffering was done to ensure that the measurement process did not affect the noise levels in the low frequency clock generator circuit.

Table 6 summarizes experiments in which the standard deviation of the period for different R1 values in the low frequency clock was measured using the statistical feature of a 500 MHz, 2 Gs/s HP Infinium 54820A oscilloscope. As can be clearly seen from the table, low frequencies lead to larger standard deviation in the period. It is reasonable to assume that the larger the period variation, the more random the resulting output bitstream and thus lower frequencies

Table 6. Low frequency clock measurements.

Frequency Hz	std(period) ns
3.74k	3.7
6.75k	8.5
13.83k	26
54.82k	42
91.91k	81

will lead to higher degrees of randomness.

5.2 NIST Test Suite

For the NIST test suite (version 1.4), all parameters were set according to the recommendations in [1] and the test sequences were 1 Mbit in size. The sample size, i.e. the number of bit sequences to pass the tests was 10. Table 7 summarizes the NIST test results for the PRNG and for the TRNG at different frequencies. The significance level α was chosen to be the default of 0.01 (99% confidence) so a test has passed if the P-value is larger than this number. Failed tests are shown in bold. It can be seen that the TRNG passes the NIST RNG test suite when the low frequency oscillator is 115 kHz or lower, corresponding to a throughput of $115/4 = 29\ kbps$. The PRNG also passes all NIST tests.

5.3 Diehard Test Suite

Although the Diehard test suite is one of the most comprehensive publically available sets of randomness tests, unfortunately there are no well-defined pass criteria. Intel calculated that the entire 250 test suite passes with a 95% confidence interval for P-values between 0.0001 and 0.9999 [9], and this method was used for our testing.

The Diehard test results are summarized in Table 8. At 36.8 kHz and 52 kHz, the test which fails is the minimum distance test. The minimum distance test is as follows, " It does this 100 times:: choose n=8000 random points in a square of side 10000. Find d, the minimum distance between the $(n^2 - n)/2$ pairs of points. If the points are truly independent uniform, then d^2, the square of the minimum distance should be (very close to) exponentially distributed with mean .995.".

It was found that for frequencies below 151 kHz (with the exception of 52 kHz and 36.8 kHz), the TRNG passes the test suite. The PRNG random sequences also passes the Diehard test.

6 Conclusion

In this paper, two FPGA based RNGs were introduced. The TRNG design demonstrates a method for producing high quality non-deterministic random numbers. Compared with previous techniques, this method does not use any special features of the FPGA and thus can be implemented on any CPLD or FPGA device. The PRNG shows that a highly secure RNG can be implemented with very small area requirements, both designs together plus the host interface occupying just 310 Virtex slices. The BBS algorithm chosen lends itself to an area efficient serial architecture, which greatly reduces circuit size, admittedly at the expense of speed. The maximum bitrate of the TRNG which could pass the NIST and Diehard test suites were 29 kbps and 4.7 kbps respectively. We recommend that the TRNG be used at or below 29 kbps, and below 4.7 kbps for very high security applications. The PRNG passes both Diehard and NIST tests and achieves an output rate of 225 bps .

Both designs were intended for embedded cryptographic applications where the random number generator is integrated on the same chip as other cryptographic hardware. Including an internal high quality random number generator may also improve security by keeping seeds and keys internal to the device.

Issues of tamper resistance have not been addressed in this work. For example, the TRNG could be made to produce a constant output by externally grounding the pin used to generate the low frequency clock. It is possible to perform rudimentary randomness tests of the TRNG output to ensure that it has not failed or been tampered with. Future work will focus on this issue.

References

[1] A. Rukhin, el. *A Statistical Test Suit For Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800-22, 2001.

[2] P. Alfke. Evolution, revolution and convolution: Recent progress in field programmable logic. *Tutorial notes*, 2001. http://www.te.rl.ac.uk/esdg/atlas-flt/talks/StockholmXilinx.pdf.

[3] B. Chetwynd, A. Elbirt, C. Paar, and W. Yip. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on VLSI*, 9(4):545–557, August 2001.

[4] D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. *Network Working Group*, RFC 1750, 1994.

[5] V. Fischer and M. Drutarovsky. True random number generator embedded in reconfigurable hardware. In *Proceedings of the Cryptographic Hardware and Embedded Systems Workshop (CHES)*, pages 415–430, 2002.

[6] S. Franco. *Design with operational amplifiers and analog integrated circuits*. McGraw-Hill, 2nd edition, 1998.

Table 7. NIST RNG test result summary for the PRNG and TRNG at different low frequency oscillator values. Failed tests are shown in bold.

Test	P-value						
	PRNG	265 kHz	151 kHz	115 kHz	52 kHz	36.8 kHz	18.8 kHz
Frequency	0.739918	**0.000439**	**0.008879**	0.739918	0.739918	0.534146	0.122325
Block Frequency	0.004301	**0.000000**	**0.000000**	0.035174	0.350485	0.534146	0.739918
Cusum-Forward	0.066882	**0.000000**	**0.000001**	0.534146	0.911413	0.350485	0.213309
Cusum-Reverse	0.350485	**0.000000**	**0.000089**	0.534146	0.739918	0.911413	0.066882
Runs	0.911413	**0.000000**	0.534146	0.350485	0.739918	0.911413	0.739918
Long Run	0.066882	0.739918	0.911413	0.739918	0.122325	0.350485	0.350485
Rank	0.739918	0.350485	0.739918	0.911413	0.911413	0.534146	0.739918
FFT	0.122325	0.534146	0.350485	0.122325	0.739918	0.350485	0.350485
Aperiodic Templates	0.350485	0.035174	0.739918	0.534146	0.350485	0.991468	0.122325
Periodic Templates	0.534146	0.213309	0.213309	0.350485	0.739918	0.122325	0.035174
Universal	0.066882	0.066882	0.350485	0.122315	0.350485	0.739918	0.122325
Approximate Entropy	0.350485	**0.000003**	0.350485	0.911413	0.534146	0.911414	0.213309
Random Excursions	0.017639	0.521333	0.981557	0.703204	0.867916	0.294149	0.017639
Serial1	0.350485	**0.000001**	0.534146	0.911413	0.122325	0.350485	0.122325
Serial2	0.213309	0.213309	0.066882	0.350485	0.534146	0.534146	0.350485
Lempel Ziv	0.066882	**0.008879**	0.213309	0.739918	0.911413	0.350485	0.534146
Linear Complexity	0.350485	0.534146	0.066882	0.739918	0.534146	0.350485	0.991468

Table 8. Diehard RNG test result summary for the PRNG and TRNG at different low frequency oscillator values. Failed tests are shown in bold.

Test	P-value						
	PRNG	265 kHz	151 kHz	115 kHz	52 kHz	36.8 kHz	18.8 kHz
Birthday Spacings	0.730825	0.224016	0.774836	0.854676	0.011842	0.457606	0.605795
Overlapping 5-Permutation	0.049365	0.512160	0.605900	0.007404	0.242983	0.892583	0.519530
Binary Rank (31x31)	0.499906	0.616320	0.723351	0.740576	0.706765	0.450976	0.264369
Binary Rank (32x32)	0.937286	0.589322	0.818277	0.452702	0.340073	0.512985	0.519530
Binary Rank (6x8)	0.946102	0.928782	0.166910	0.992197	0.558210	0.652454	0.264369
Bitstream	0.05040	1.00000	0.43382	0.83427	0.54249	0.17241	0.20124
OPSO	0.7242	0.8957	0.9886	0.6214	0.7938	0.7311	0.9808
OQSO	0.4403	0.4470	0.2539	0.7497	0.3784	0.3875	0.0952
DNA	0.6235	0.4271	0.1613	0.9202	0.5149	0.7231	0.3713
Steam Count-the-1	0.895581	0.998743	0.801343	0.418211	0.987518	0.225811	0.915539
Byte Count-the-1	0.373290	0.600272	0.646492	0.599296	0.992857	0.872483	0.395120
parking Lot	0.089494	0.322808	0.246694	0.863809	0.585747	0.287980	0.246216
Min. Distance	0.841020	0.684397	0.232250	0.541518	**0.999991**	**1.00000**	0.994580
3D Spheres	0.611194	0.131376	0.332614	0.668792	0.891787	0.093055	0.928373
Squeeze	0.954492	0.738270	0.268667	0.147953	0.801557	0.311240	0.426530
Overlapping Sums	0.855183	0.691566	0.469958	0.926731	0.669721	0.06766	0.436228
Runs up	0.045029	0.743994	0.679103	0.479088	0.381046	0.607365	0.587124
Runs down	0.473325	0.289460	0.350040	0.328526	0.095483	0.632704	0.837897
Craps	0.668026	0.843023	0.330808	0.106158	0.719738	0.233184	0.310008

[7] A. Gerosa, R. Bernardini, and S. Pietri. A fully integrated 8-bit, 20MHz, truly random numbers generator, based on a chaotic system. In *SSMSD. 2001 Southwest Symposium on Mixed-Signal Design*, pages 87–92, 2001.

[8] P. Hortensius, R. McLeod, and H. Card. Parallel random number generation for VLSI systems using cellular automata. *IEEE Transactions on Computers*, 38(10):1466–1473, Oct. 1989.

[9] Intel Platform Security Division. The intel random number generator. *Intel technical brief*, 1999. ftp://download.intel.com/design/security/rng/techbrief.pdf.

[10] B. Jun and P. Kocher. The intel random number generator. *White paper by Cryptographic Research Inc.*, 1999. ftp://download.intel.com/design/security/rng/CRIwp.pdf.

[11] D. Knuth. *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms*. Addison-Wesley, 1981.

[12] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on computing*, 15(2):364–383, 1986.

[13] A. Lenstra and H. Lenstra Jr, editors. *The development of the number field sieve*, volume 1554. Lecture Notes in Mathematics, Springer-Verlag, 1993.

[14] G. Marsaglia. DIEHARD: a battery of tests for random number generators. 2002. http://stat.fsu.edu/∼geo/diehard.html.

[15] P. Martin. An analysis of random number generators for A hardware implementation of genetic programming using FPGAs and Handel-C. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 837–844, 2002.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 5th edition, 2001.

[17] O.Y.H. Cheung, K.H. Tsoi, K.H. Leung, P.H.W. Leong, and M.P. Leong. Tradeoffs in parallel and serial implementations of the international data encryption algorithm IDEA. In *Proceedings of the Cryptographic Hardware and Embedded Systems Workshop (CHES)*, pages 333–347. LNCS 2162, Springer, 2001.

[18] V. Pasham and S. Trimberger. *High-Speed DES and Triple DES Encryptor/Decryptor*. Xilinx, Inc., 2001. Applications Note XAPP270.

[19] C. Petrie and J. Connelly. A noise-based IC random number generator for applications in cryptography. *IEEE Journal of Solid State Circuits*, 47(5):615–621, 2000.

[20] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, M.Y. Wong, and K.H. Lee. Pilchard – a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001 (to appear).

[21] R. Ramaswamy. Application of a key generation and distribution algorithm for secure communication in open systems interconnection architecture. In *Security Technology, 1989. Proceedings. 1989 International Carnahan Conference on, 1989*, pages 175–180, 1989.

[22] R.C. Fairfield, R.L. Mortenson, and K.B. Coulthart. An LSI Random Number Generator (RNG). In *Advances in Cryp-*

tography: Proceedings of Crypto 84, pages 203–230. LNCS 0196, Springer-Verlag, 1984.

[23] RSA Labs. *FAQ*, 2000. http://www.rsasecurity.com/rsalabs/faq/index.html.

[24] B. Schneider. *Applied Cryptography*. John Wiley & Sons, second edition, 1996.

[25] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider. FPGA implementation of neighborhood-of-four cellular automata random number generators. Technical report, HP Labs Technical Reports, 2001.

[26] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *Proceedings., 11th Symposium on Computer Arithmetic*, pages 252 –259, 1993.

[27] T. Stojanovski, J. Pil, and L. Kocarev. Chaos-based random number generators. Part II: practical realization. *IEEE Transactions on Circuits and Systems – I: fundamental Theory and Application*, 48(3):382–385, March 2001.

[28] T. Rinne, T. Ylonen, Tero Kivinen, and M Saarinen Sami. *SSH Authentication Protocol*. Network Working Group, Internet Draft, Internet Engineering Task Force (IETF), 2002.

[29] U. Vazirani and V. Vazirani. Efficient and secure pseudo-random number generation. In *Proc. 25th IEEE Symp. on the Foundations of Comput. Sci.*, pages 458–463, 1984.

[30] R. K. Watkins, J. C. Isaacs, and S. Y. Foo. Evolvable random number generators: A schemata-based approach. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 469–473, 2001.

[31] S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123 – 169, 1986.

[32] Xilinx, Inc. *Xilinx Libraries Guide*, 1999.