AN-1200 True Random Number Generator Hardware

In cryptographic applications such as key generation and signing, there is a need for random numbers. There are many methods which hardware or firmware can use to generate pseudo-random numbers. However, pseudo-random number generators are deterministic: if the internal state of the generator is known, then the output of the generator is entirely predictable. This makes pseudo-random number generators inappropriate for cryptographic applications. These applications require a true random number generator (TRNG).

A TRNG uses unpredictable physical phenomena to generate random numbers. An example of a TRNG would be to amplify thermal noise from a resistor, then use an analog-to-digital convertor to convert this noise into numbers. Such a system usually requires a lot of discrete components due to the large gain needed. Because thermal noise is the result of chaotic microscopic effects, this generator will generate unpredictable numbers. This unpredictability is referred to as entropy. Entropy can be measured in bits, for example, a fair coin toss will produce 1 bit of entropy, as there are two equally possible outcomes. Independent sources of entropy add, so tossing 8 independent, fair coins will produce 8 bits of entropy. A biased coin that always lands on heads will produce 0 bits of entropy, since a fully biased coin is completely predictable.

A good TRNG will be able to quickly generate enough bits of entropy, so that cryptographic keys or secrets cannot be feasibly guessed. A bad random number generator can compromise a cryptographic system. For example, a faulty random number generator was the cause of several thefts of the cryptocurrency Bitcoin [1].

Entropy source

The true random number generator described in this application note uses free-running ring oscillators. These ring oscillators are constructed using an odd number of inverters constructed out of LUTs. The oscillators are free-running in the sense that they are not locked to other clocks, and will accumulate jitter – see Figure 1 for a frequency-domain representation of the jitter from a single oscillator. This jitter is a result of many unpredictable effects, including thermal noise, so a generator constructed out of ring oscillators is justified in being called a true random number generator.

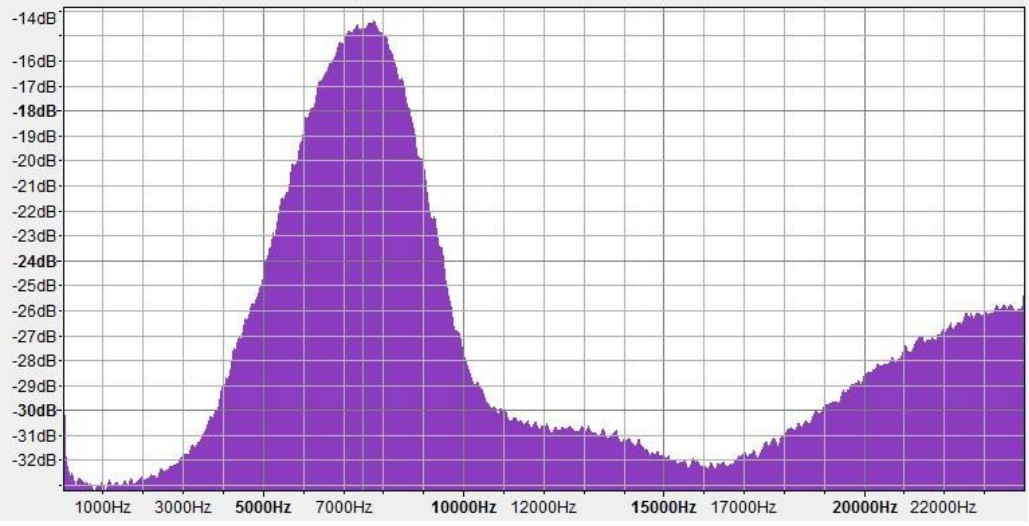


Figure 1. FFT of the output of a single ring oscillator implemented in a GreenPAK™ device. This ring oscillator oscillates at a nominal frequency of about 19 MHz, and the output has been digitally sampled at 48 kbit/s. Because the oscillator's frequency is far above the Nyquist frequency, the oscillator signal has been aliased to ~7.5 kHz. The jitter in this ring oscillator is visible, as the peak is not infinitely thin, as would be the case for an ideal, jitter-free oscillator. Instead, the peak has a 3 dB bandwidth of about 2 kHz.

To increase the amount of entropy available, and to make the design more robust, many ring oscillators should be used. Because the oscillators are independent, they will randomly drift in and out of phase of each other. The oscillators can be sampled by periodically taking the XOR of all their outputs, which represents the oscillators' relative phase. This is not the most efficient way to sample oscillators, as many oscillations are required before the oscillators will drift out of phase with each other. However, it is a simple method that does not require the use of many ring oscillators – a higher-throughput design could require over 100 oscillators [2].

Whitening

A perfect entropy source generates a stream of bits which have a white power spectrum (equal power in all frequencies; the spectrum is flat) and which are statistically unbiased (nearly equal ones and zeroes). Real entropy sources are imperfect – they generate a bitstream which is statistically poor in some way. For example, noise in circuits often has a power spectrum which is not white. Indeed, Figure 1 shows that the ring oscillators used in this application note have a noise power spectrum which is not flat. Asymmetries in a circuit (e.g. due to a difference in rising and falling propagation delays) could cause the output to become biased. Statistically imperfect entropy sources are still useful, but they require post-processing. This post-processing is referred to as whitening.

To illustrate the need for whitening, imagine that someone wishes to extract entropy from a series of coin tosses. They are using a hypothetical set of 16 coins – 8 of which are fair, 8 of which are completely biased, and will always land on heads. For the sake of analogy, this hypothetical person is also unable to distinguish the fair coins from the biased coins. Flipping all 16 of these coins will lead to a partially predictable result, as there are guaranteed to be at least 8 heads appearing. However, there should still be a total of 8 bits of entropy. A whitener can be used to take the results of the 16 coin tosses, and process these results to obtain 8 random bits.

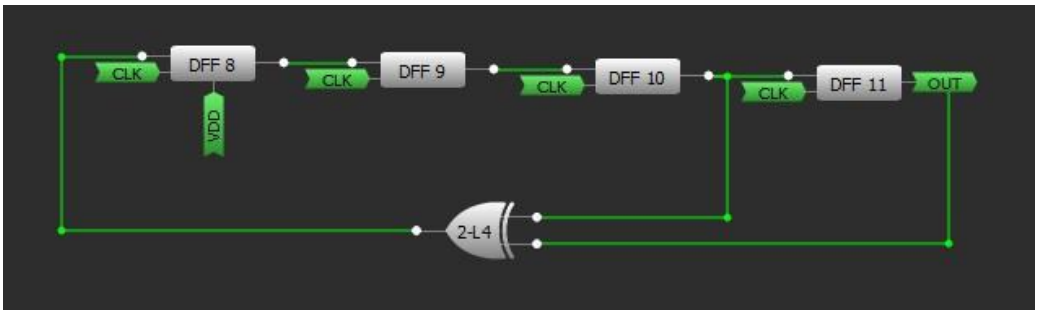


Figure 2. Example of a 4-bit LFSR, implemented using 4 D flip flops and a XOR gate (this particular LFSR is only an example; it is not used in the TRNG implementation). This LFSR will produce the following output: 111100010011010111100010011010111100010011010111 10. Inspection of this pseudo-random bitstream reveals that it repeats every 15 bits.

The whitener used in this application note is a linear feedback shift register (LFSR). A LFSR is easy to construct in hardware and consists of a shift register, with some of the register contents XORed together and fed back into the input of the shift register – see Figure 2 for an example of this. On its own, the output of a LFSR will be a repeating pseudo-random sequence of bits. Careful choice of what registers to XOR together means that the cycle length of this pseudo-random sequence will be of maximal cycle length. The maximum possible cycle length of a n-bit shift register is $2^n - 1$. A table of the suitable XOR choices is given in [3]. A large, maximal LFSR will produce a bitstream which is unbiased and white.

A LFSR will produce a bitstream with good statistical properties, but on its own, it is still only pseudo-random. For this application note, the feedback of the LFSR is also XORed with the raw output of the ring oscillators. The LFSR's internal state will now be influenced by the ring oscillators, converting the LFSR from a pseudo-random number generator into a true random number generator. Even with this modification, the overall feedback structure of a large, maximal LFSR ensures that its output is still unbiased and white. Hence an LFSR, used in this way, fulfils the function of a whitener.

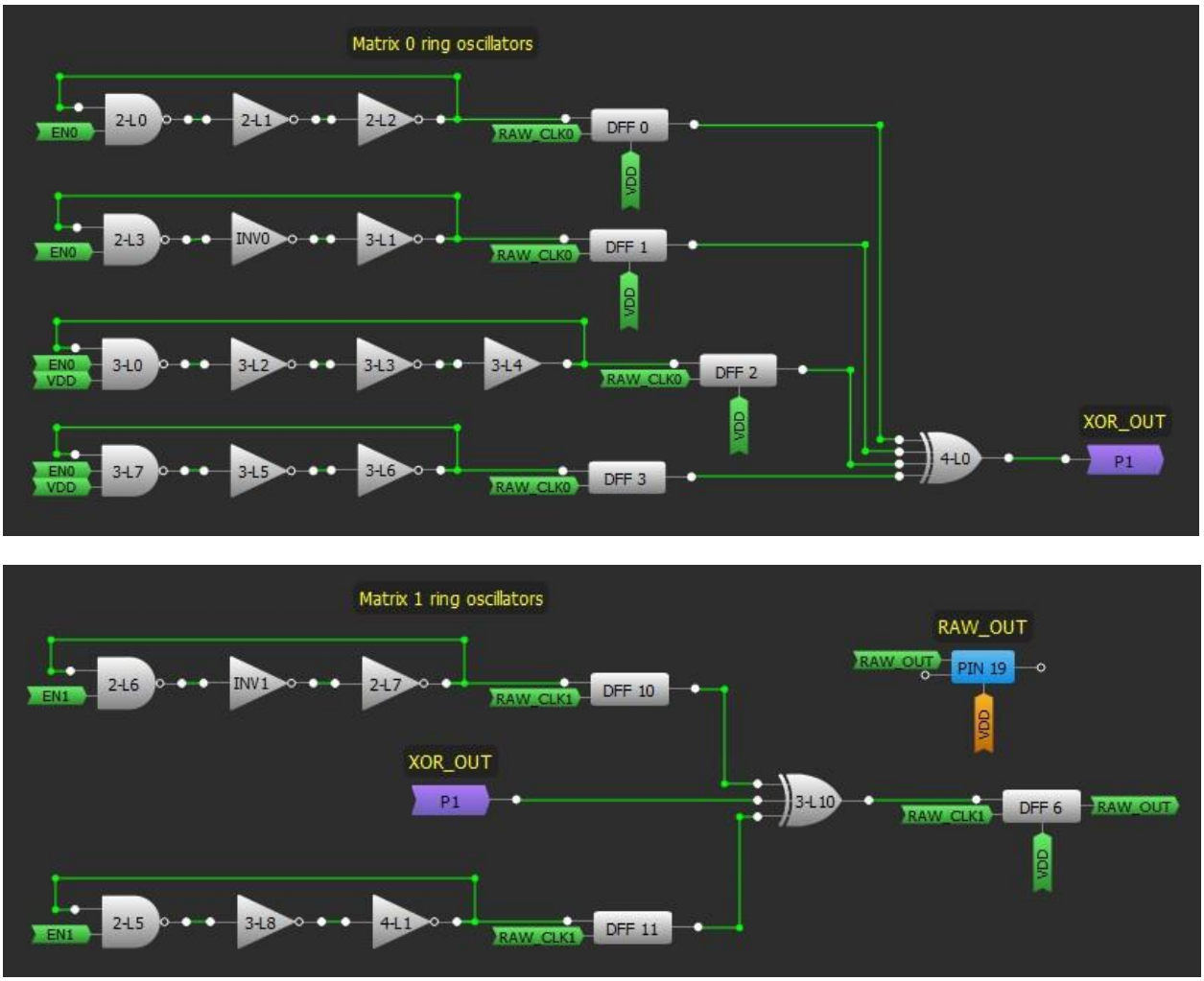


Figure 3. Ring oscillator implementations. The top image shows the four oscillators implemented on the matrix 0 side. The bottom image shows the two oscillators implemented on the matrix 1 side.

There are enough LUTs in the SLG46620V for six ring oscillators. Their implementation is shown in Figure 3. Each ring oscillator starts with a NAND gate instead of a simple inverter. Connected to each NAND gate is a global enable signal (net EN0 for matrix 0, net EN1 for matrix 1). When this enable signal is low, all the NAND gates will output high, regardless of the state of the other input. This will cause the ring oscillators to stop oscillating and enter a static, well-defined state. When this enable signal is high, the NAND gate effectively functions as another inverter.

All the ring oscillators are constructed out of 3 inverters, except for one of the rings, which has an extra buffer. All the rings are constructed out of unique combinations of 2-bit LUTs, 3-bit LUTs, 4-bit LUTs, and the INV0/INV1 blocks. This is a crucial design choice. Experimentation has revealed that if any two rings are constructed out of the same types of blocks, they will have very similar oscillation frequencies. The frequencies are so similar, that small amounts of crosstalk between the oscillators (probably a result of switching noise) will eventually cause the oscillators to phase-lock together. This phenomenon (generally referred to as injection locking) will cause a catastrophic failure in entropy generation, as the oscillators will constantly be exactly in phase with each other. 2-bit LUTs, 3-bit LUTs, 4-bit LUTs, and the INV0/INV1 blocks all have different propagation delays, so using unique combinations of these types of blocks guarantees that the oscillator frequencies will be different.

The output of each ring oscillator is latched on the rising edge of the RAW_CLK signal. This follows the advice of [4], where it was observed that failing to latch oscillator outputs causes setup and hold time violations on the input to XOR gates. This approach also helps to reduce switching noise,

minimizing the amount of harmful crosstalk between oscillators.

Each ring oscillator was measured to have a spectrum like that shown in Figure 1, with a noise bandwidth of about 2 kHz. The rate of entropy can be calculated using Hartley's law:

$$C = 2B \log_2 M$$

where C is the bitrate, B is the bandwidth and M is the number of distinguishable states. Using B = 2 kHz, and M = 2 (since the output of a ring oscillator is either a 0 or a 1) results in a bitrate of 4 kbit/s. Hence each ring oscillator contributes 4 kbit/s of entropy.

Finally, the latched outputs of all oscillators are XORed together, to measure their relative phase. This XOR output is latched on the rising edge of the RAW_CLK signal, to ensure that that there are no spurious transitions in the output signal. This results in the RAW_OUT signal. RAW_OUT is sent out of pin 19 for the user. The choice of pin 19 for RAW_OUT was completely arbitrary; system designers are free to re-assign pins to make PCB routing easier.

Realization with GreenPAK designer: LFSR

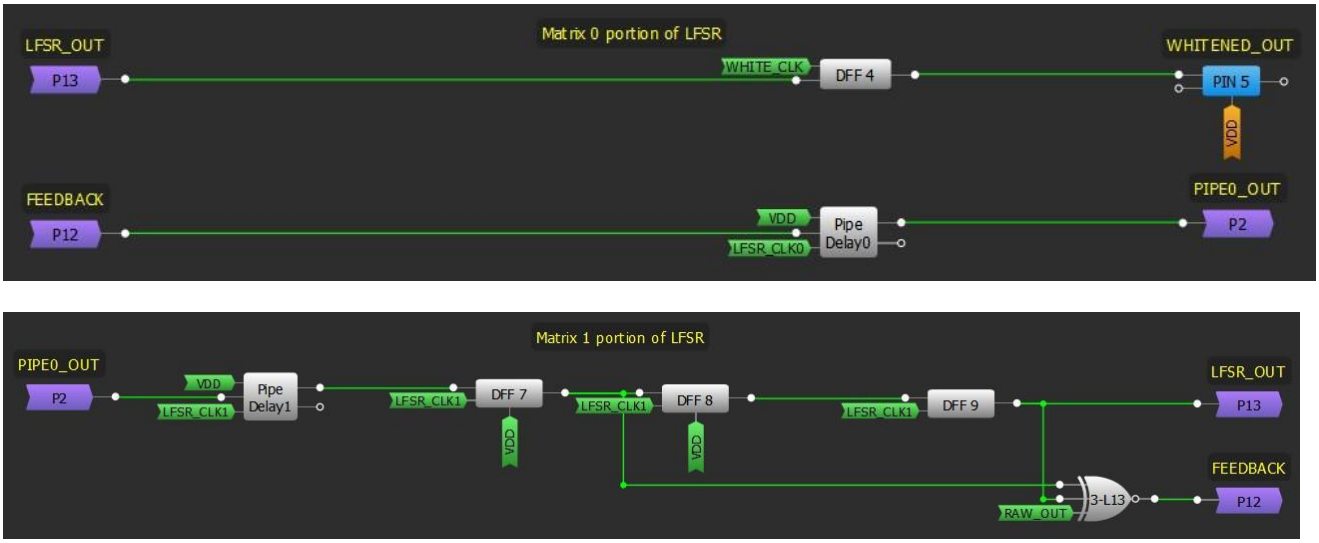


Figure 4. LFSR implementation.
The top image shows the portion of the LFSR in matrix 0. The bottom image shows the portion of the LFSR in matrix 1. It was necessary to split the LFSR across matrices to fully use both pipe delay blocks.

A 35-bit LFSR was implemented, because there exists a maximal 35-bit LFSR with a very simple structure: the feedback consists of the XOR of registers 33 and 35 (see the n=35 entry of Table 3 in [3]). The LFSR uses both pipe delay blocks in the SLG46620V, where each pipe delay block consists of 16 D flip flops in serial. Pipe delay 0 contributes the first 16 bits of the shift register, pipe delay 1 contributes the next 16 bits, and D flip flops 7, 8, and 9 form the last three bits of the shift register. The resulting 35-bit LFSR has a cycle period of 235 – 1, which is extremely long compared to the LFSR clock rate.

The feedback element shown in Figure 4 uses NXOR instead of XOR. This does not change the structure of the LFSR, or the cycle period. However, it does ensure that the LFSR does not enter a lock-up state (where cycle period = 1) when the shift register is initialized to an all-0s state, which is the case during power-on.

The LFSR is clocked/shifted by the global LFSR_CLK signal (net LFSR_CLK0 for matrix 0, net LFSR_CLK1 for matrix 1). The output of the LFSR (LFSR_OUT) is not directly exposed to the user. Instead, the output of the LFSR is latched on the rising edge of the WHITE_CLK signal. Note that WHITE_CLK runs at a slower rate than LFSR_CLK – this is to allow some bits of LFSR_OUT to be discarded. This latched output is sent to pin 5, as the WHITENED_OUT signal. Like with RAW_OUT, the choice of pin 5 was completely arbitrary.

Realization with GreenPAK designer: clocks

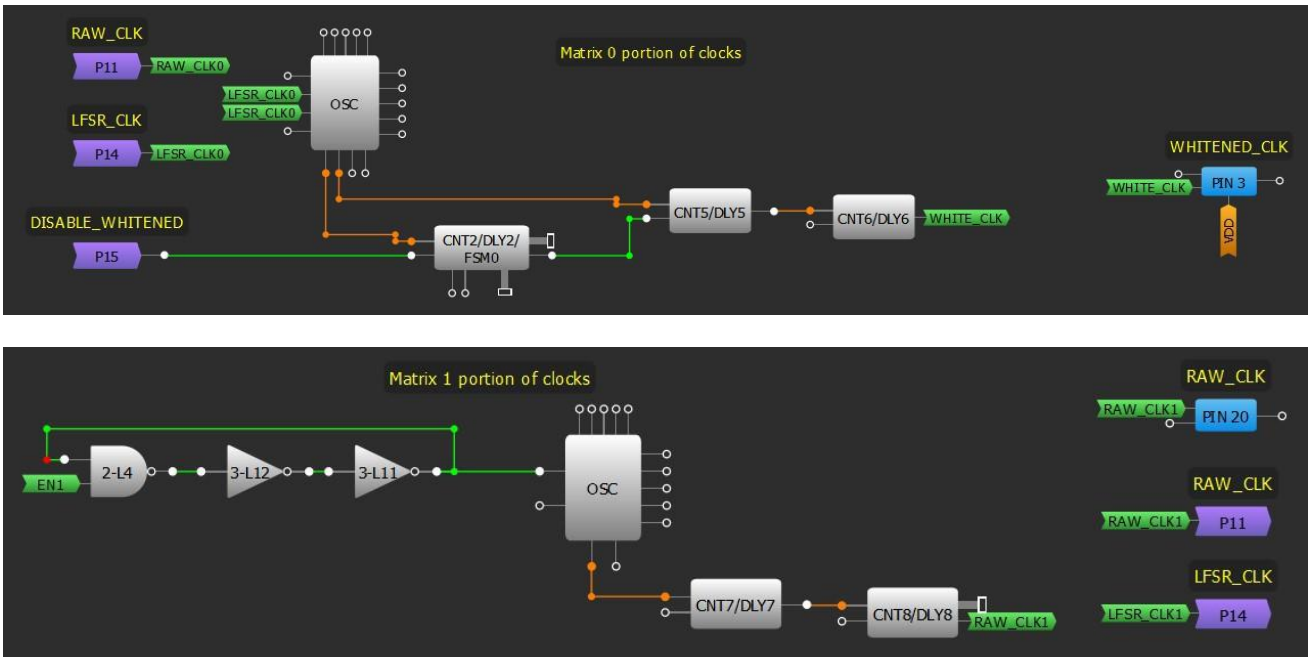


Figure 5. Clock structure of TRNG. The top image shows the portion in matrix 0. The bottom image shows the portion in matrix 1.

The master clock source for the TRNG is a ring oscillator constructed out of LUTs, visible in the bottom image of Figure 5. Although the SLG46620V does have many in-built clocks, none of those were used. The reason for this was to improve rejection against common-mode interference such as power supply noise, temperature variations, and power supply voltage variations. By constructing the master clock in the same way as the entropy source, these common-mode interference sources should affect both the master clock and the entropy source in the same way, partially cancelling out the effect of the interference.

The master clock is fed into matrix 1's EXT. CLK2, where CNT7 and CNT8 are set up as counters to divide the master clock by 306, to achieve a nominal (at VDD = 3.3 V) RAW_CLK of about 80 kHz. CNT7 and CNT8 are set up in this particular cascade (CNT7 has counter data = 152, CNT8 has counter data = 1) to ensure that RAW_CLK has a duty cycle of 50%. RAW_CLK is used as the sampling clock for the ring oscillators, hence RAW_OUT will have a nominal bitrate of 80 kbit/s. RAW_CLK is also output to pin 20 (this pin assignment is arbitrary) so that the user can sample RAW_OUT at the right times.

LFSR_CLK has the same frequency as RAW_CLK – the only difference is that LFSR_CLK can be suppressed by asserting ENABLE_WHITENED low. LFSR_CLK is used to clock the shift registers in the LFSR. In matrix 0, LFSR_CLK is fed into EXT. CLK1, so that CNT5 and CNT6 can derive WHITE_CLK from it. CNT5 and CNT6 are set up as counters, both with counter data = 1, so they divide LFSR_CLK by 4. The divided clock, WHITE_CLK, is used to decimate the LFSR output. WHITE_CLK is sent to pin 3 (this pin assignment is arbitrary) so that the user can sample WHITENED_OUT at the right times. WHITE_CLK has a nominal frequency of 20 kHz; WHITENED_OUT has a bitrate of 20 kbit/s.

The final bitrate of 20 kbit/s was chosen because testing revealed that each ring oscillator could provide about 4 kbit/s of entropy. Since the entropy source consists of six ring oscillators (see Figure 3), and assuming the entropy sources add, there should be a total of about 24 kbit/s of entropy in RAW_OUT. This is derated to 20 kbit/s to account for imperfections in the ring oscillators and LFSR.

Note that while RAW_OUT has a nominal bitrate of 80 kbit/s, the expected entropy rate of RAW_OUT is only 20 kbit/s. This is because RAW_OUT is expected to have statistical imperfections

which make its bitstream partially predictable. On the other hand, WHITENED_OUT is completely unpredictable, hence it has a nominal bitrate (of 20 kbit/s) equal to its expected entropy rate.

The output of DLY2 holds CNT5 in reset; the DISABLE_WHITENED signal halts WHITE_CLK. DLY2 is set up so that the falling edge of the DISABLE_WHITENED signal is delayed by 400 counts of LFSR_CLK. The effect of this is that after the whitened output is enabled, the first 400 bits of the LFSR will be discarded. It is necessary to discard the first 400 bits of LFSR output so that fresh entropy can mix into the LFSR's internal state.

Realization with GreenPAK designer: enable logic

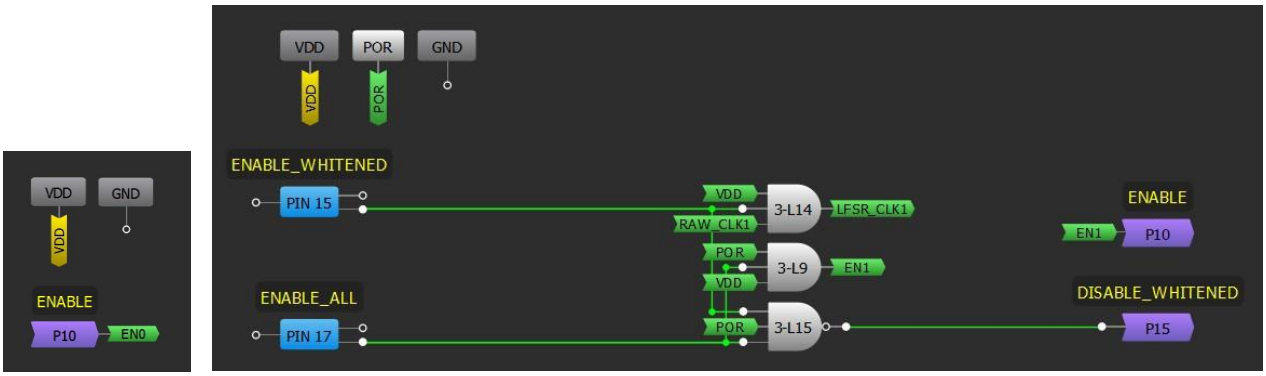


Figure 6. Enable logic of TRNG.
The left image shows the portion in matrix 0. The right image shows the portion in matrix 1.

The remaining configuration is dedicated to enable logic. The enable logic allows the user to save power by disabling sections of the TRNG. The ENABLE_ALL signal is connected to pin 17 (this pin assignment is arbitrary). If the user asserts pin 17 low, then all ring oscillators will stop oscillating. This will cause the entire TRNG to enter a static state.

The ENABLE_ALL signal is also ANDed with the output of the POR block. This is necessary, as the ring oscillators will not begin oscillating unless they begin from a well-defined state. The POR clock is configured to output high 4 µs after power-on. This gives the ring oscillators 4 µs to settle into the well-defined disabled state (see the section "Realization with GreenPAK designer: ring oscillators"), ensuring that the ring oscillators will reliably start oscillating after power-on.

The ENABLE_WHITENED signal is connected to pin 15 (this pin assignment is arbitrary). If the user asserts pin 15 low, then LFSR_CLK will be disabled, and the entire LFSR section will enter a low-power static state. A related signal, DISABLE_WHITENED, is used to suppress the first 100 bits of whitened output (i.e. the first 400 bits of LFSR output), so that fresh entropy is mixed into the LFSR internal state before anything is output. For this reason, DISABLE_WHITENED will be asserted high whenever the LFSR or entropy source is (re)started.

Test results

The TRNG described in this application note was implemented in a SLG46620V, and samples were taken from the RAW_OUT and WHITENED_OUT pins. This was done using a LPC11U24 microcontroller that was set up to sample RAW_OUT/WHITENED_OUT on the falling edge of RAW_CLK/WHITENED_CLK. These samples were sent to a PC over a UART-to-USB converter, and the samples were logged into a file for later analysis.

The result of an FFT performed on the raw output is shown in Figure 7. While the spectrum is approximately flat, to within 2.5 dB, there is clearly a structure of peaks and troughs. This structure is inconsistent with the expected result of a perfect entropy source (a flat spectrum).

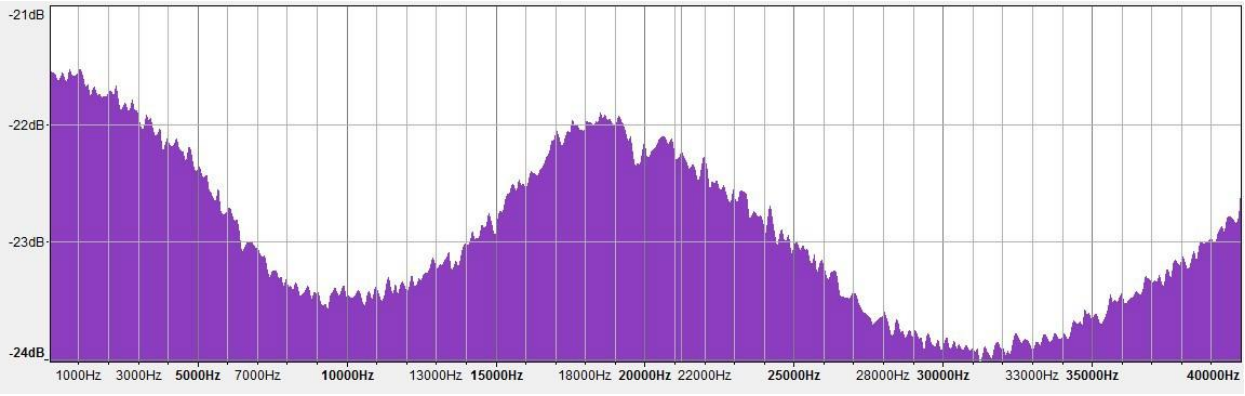


Figure 7. FFT results of about 3 million bits of raw output (sampled from RAW_OUT).
Wide peaks and troughs in the spectra are clearly visible.

The result of an FFT performed on the whitened output is shown in Figure 8. This spectrum is flat, with the (statistical) variance observed to be within 0.4 dB.

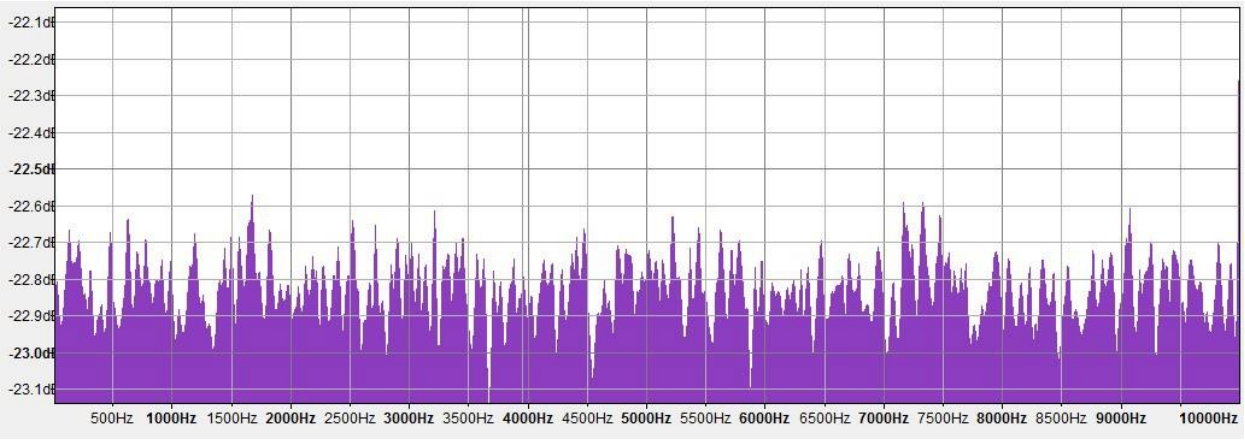


Figure 8. FFT results of about 3 million bits of whitened output (sampled from WHITENED_OUT).
The spectrum is flat (white).
The observed variance is statistical in nature.

Further statistical testing can be done using a suite of statistical tests developed by NIST, described in detail in [5]. For a brief explanation of what each test is examining, see [8]. The NIST tests are specifically adapted to testing random bitstreams. For example, the "Frequency" test examines whether the proportion of ones and zeroes is the same i.e. the "Frequency" test checks that the bitstream is unbiased. To test the TRNG, about 80,000,000 bits of RAW_OUT and WHITENED_OUT were collected and recorded in files. These files were split into 100 bitstreams of 800,000 bits each, and then analysed, using the assess tool described in [5]. Appendix A contains the final results ("finalAnalysisReport.txt") for the raw output, and Appendix B contains the final results for the whitened output.

An accessible introduction to interpreting the NIST test results is given in [7]. The NIST test results contain a lot of columns, but the important ones are the "P-VALUE" and "PROPORTION" columns. Values in the "P-VALUE" column should be uniformly distributed between 0 and 1. Values close to 0 (e.g. 0.000000) or 1 (e.g. 0.999999) are indicative of failure. The "PROPORTION" column describes how many of the 100 bitstreams passed the statistical tests. Statistically, some of the bitstreams are expected to fail. The minimum pass rate is described at the bottom of each appendix. Asterisks in either the "P-VALUE" or "PROPORTION" columns are indicative of failure.

The general conclusion from the NIST test results is that the raw output fails most of the statistical tests. This failure is expected – the raw output is an imperfect source of entropy. On the other hand, the whitened output passes every statistical test. This indicates that the whitened output can be used as a source of almost-perfectly random bits.

How to use the TRNG

The TRNG implemented in this application note requires only the SLG46620V to generate the random bitstream; it does not require any external components, except for decoupling capacitors for the SLG46620V. The raw output can be sampled by connecting RAW_CLK (pin 20) and RAW_OUT (pin 19) to a microcontroller – see Figure 9 for an example of what these signals look like.

If the microcontroller supports SPI in slave mode, then RAW_CLK can be connected to SCLK and RAW_OUT can be connected to MOSI, with CPOL set to 0 and CPHA set to 1 (SPI mode 1). Alternatively, the microcontroller can be set up to interrupt on the falling edge of RAW_CLK, with the interrupt handler sampling RAW_OUT.

The whitened output can be similarly sampled using the WHITENED_CLK (pin 3) and WHITENED_OUT (pin 5) signals – see Figure 9 for an example of what these signals looks like. The whitened output is the most appropriate output for most applications.

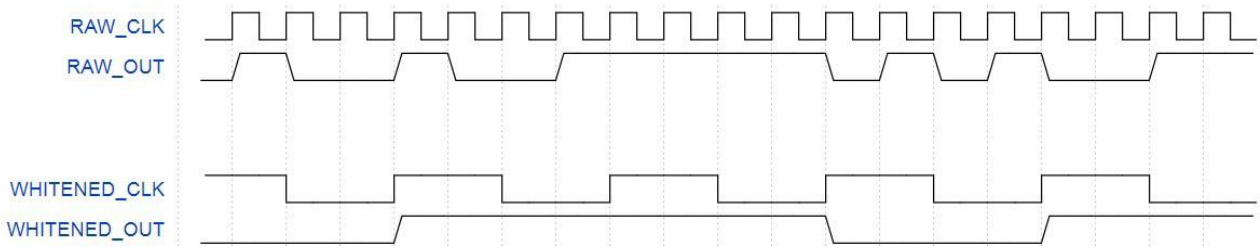


Figure 9. Timing diagram showing an example of what the output signals look like. RAW_OUT transitions on the rising edge of RAW_CLK; WHITENED_OUT transitions on the rising edge of WHITENED_CLK. Both RAW_OUT and WHITENED_OUT should be sampled on the falling edge of their respective clocks. In this example, RAW_OUT is depicted as outputting the bitstream 1001001111101010011, and WHITENED_OUT is depicted as outputting the bitstream 01101. RAW_CLK runs at a nominal rate of 80 kHz; WHITENED_CLOCK runs at a quarter of RAW_CLK, at a nominal rate of 20 kHz.

The whitened output can be disabled by asserting ENABLE_WHITENED (pin 15) low. The entire TRNG can be disabled by asserting ENABLE_ALL (pin 17) low. Both enable pins are pulled-up, so if they are both left unconnected, all sections of the TRNG will be enabled.

A summary of signals is given in Table 1.

Name	Pin	Type	Description
RAW_OUT	19	Digital push-pull output	Raw output of the TRNG – the latched output of the XOR of all ring oscillators, representing their relative phase.
RAW_CLK	20	Digital push-pull output	Sampling clock for RAW_OUT. Runs at a nominal 80 kHz. RAW_OUT transitions on the rising edge of this clock.
WHITENED_OUT	5	Digital push-pull output	Output of the TRNG after whitening by a LFSR.
WHITENED_CLK	3	Digital push-pull output	Sampling clock for WHITENED_OUT. Runs at a nominal 20 kHz. WHITENED_OUT transitions on the rising edge of this clock.
ENABLE_ALL	17	Digital input with pull-up	When low, stops all clocks within the TRNG, putting it into a low-power static state; RAW_CLK and WHITENED_CLK will stop ticking.
ENABLE_WHITENED	15	Digital input with pull-up	When low, stops the LFSR clock, causing the whitened output to halt; WHITENED_CLK will stop ticking.

Table 1. Summary of TRNG signals

Security considerations

High-security applications should verify that the entropy source is working properly. To do this, they must not test the whitened output; the whitened output should not be trusted to be truly random, even if it passes statistical tests. This is because the maximal 35-bit LFSR used in this application note is a good pseudo-random number generator, and will produce results that will pass most statistical tests, even if the entropy source is completely broken. Instead, applications should try to detect failure by running statistical tests on the raw output. For example, applications could test for unusually long strings of 0s or 1s, or use a FFT to test if the raw output spectrum is less flat than Figure 7.

For simplicity, the whitened output can be used directly in cryptographic operations. Applications demanding greater security should use the whitened output to seed a cryptographically-secure pseudo-random number generator (CSPRNG), continually reseeding the CSPRNG as new whitened output bits become available. This approach has the advantages:

- The CSPRNG can be used to generate a high throughput of on-demand bits; the application won't be limited to the 20 kbit/s produced by this TRNG.
- The reseeding operation allows entropy to accumulate in the CSPRNG's internal state. This guarantees that the CSPRNG will become increasingly unpredictable (and hence secure) over time, even in the event of partial failure of the TRNG's entropy source.
- Other entropy sources (e.g. data from radio receivers) can be used to reseed the CSPRNG. This will make the cryptographic system more robust against failure of an entropy source.

A cryptographic system using a SLG46620V as a TRNG is trivially vulnerable to invasive attacks. An attacker who has physical access to the system can simply remove the SLG46620V, replacing it with a device that produces a completely predictable bitstream. This can be countered by making physical access difficult, or attempting to detect physical access.

TRNGs based on ring oscillators are known to be vulnerable to signals injected to their power supply [6]. This can be countered by filtering the power supply. A possible extension to this application note is to use the ACMP and VREF blocks to detect such injection attacks.

Downsizing the design

The TRNG design described in this application note can be implemented in a smaller GreenPAK, to reduce costs and board space requirements. The number of ring oscillators used in the entropy source could be reduced, down to a minimum of two. Since there are three times fewer oscillators being used, this would reduce the throughput of the design to a third: approximately 6.6 kbit/s.

D flip flop requirements could be reduced by using a smaller LFSR for a whitener. The minimum LFSR size depends on the number of ring oscillators, and on the oscillator sampling clock rate. The minimum LFSR size can be experimentally determined by using successively smaller LFSR lengths, until the whitened output begins to fail the NIST statistical tests.

Usually, a TRNG is used in a system that contains a microcontroller. Thus, a minimal GreenPAK design could omit the whitener completely, relying on the microcontroller to perform whitening. The firmware in that microcontroller could easily implement a very large LFSR, or it could use a cryptographic hash function to implement a more robust whitener.

Conclusion

GreenPAK can be used to implement a compact true random number generator that requires no external components (except for decoupling capacitors), and generates true random numbers at a rate of 20 kbit/s. The TRNG has a simple interface, and has power-saving features, making it ideal for mobile, space-constrained devices that require an entropy source for cryptographic operations.

References

↑ 1. D. Gilson, "Blockchain.info issues refunds to bitcoin theft victims," Aug. 21, 2013. [Online]. Available: http://www.coindesk.com/blockchain-info-issues-refunds-to-bitcoin-theft-victims/. [Accessed: Jun. 30, 2017].

↑ 2. B. Sunar, W. J. Martin, and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks", *IEEE Trans. Comput.*, vol. 56, no. 1, Jan. 2007.

↑ 3. P. Alfke, "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators," Xilinx Application Note, XAPP 052, Jul. 7, 1996.

↑ 4. K. Wold, and C. H. Tan, "Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings," *Int. J. of Reconfigurable Computing*, vol. 2009, article ID 501672, 2009.

↑ 5. A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," NIST Special Publication 800-22, revision 1a, Apr. 2010.

↑ 6. A. T. Markettos, and S. W. Moore, "The Frequency Injection Attack on Ring-Oscillator-Based True Random Number Generators," in *Proc. 11th Int. Workshop Cryptographic Hardware and Embedded Syst.*, Lausanne, Switzerland, 2009, pp. 317-331.

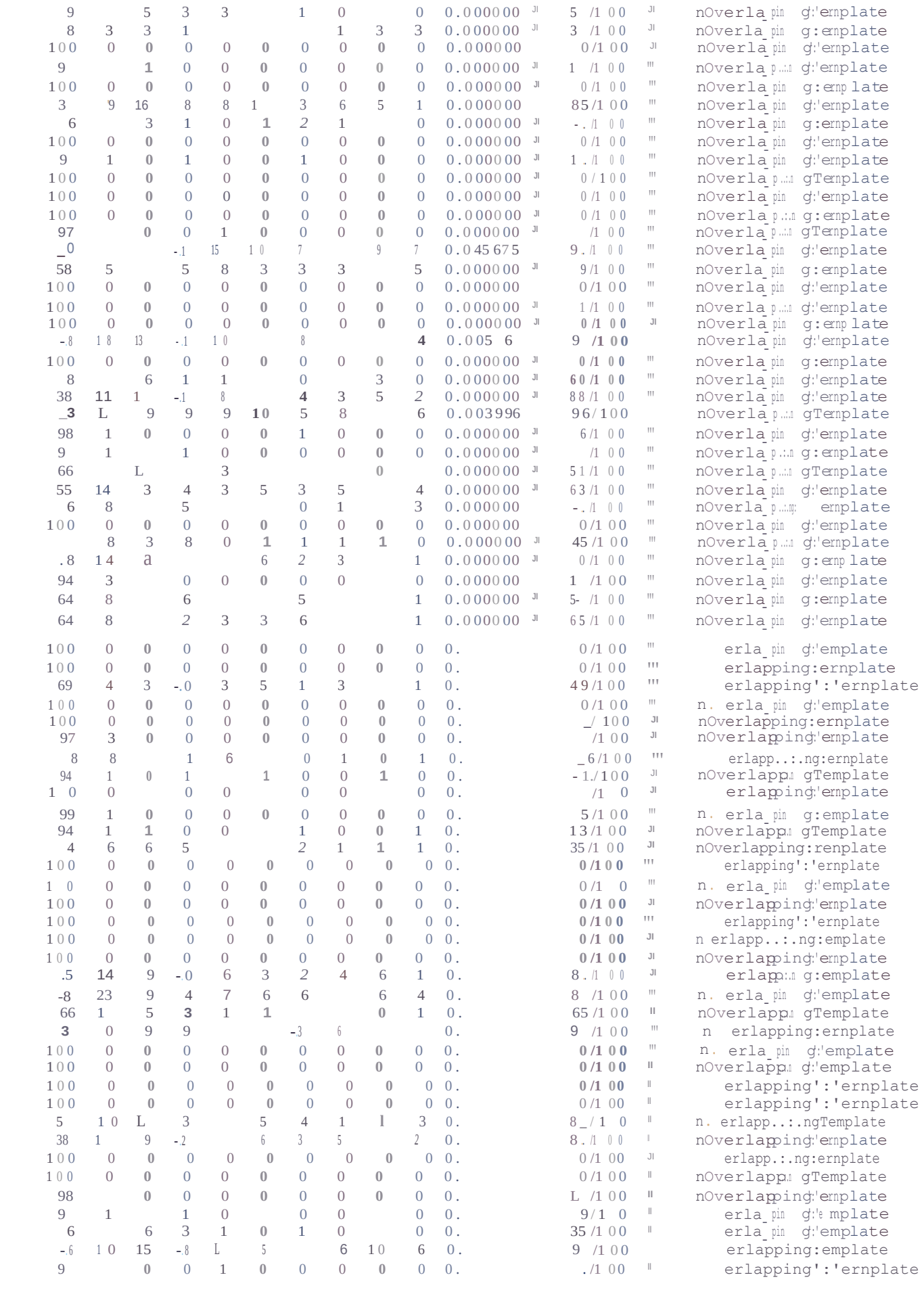
↑ 7. "Interpretation of the results of NIST (p)NRG suite," Jan. 9, 2017. [Online]. Available: https://crypto.stackexchange.com/questions/19861/interpretation-of-the-results-of-nist-pnrg-suite. [Accessed: Jul. 16 2017].

↑ 8. "Guide to the statistical tests," Jul. 16, 2014. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/stats_tests.html. [Accessed: Jul. 17 2017].

Appendix A: NIST SP 800-22 test results for raw output

For testing methodology and interpretation of results, see the “Test results” section.

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <lots_raw_out_80.0k.bin>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  Frequency
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  BlockFrequency
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  CumulativeSums
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  CumulativeSums
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  Runs
 63  13   4   5   4   4   3   2   1   1 0.000000 *   59/100  *  LongestRun
  6  10  11   9   7  10  12   9  13  13 0.834308     99/100     Rank
 52  10   9   5   2   3   6   4   2   7 0.000000 *   69/100  *  FFT
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 99   1   0   0   0   0   0   0   0   0 0.000000 *    6/100  *  NonOverlappingTemplate
 37  15  13   5  10   6   5   6   2   1 0.000000 *   84/100  *  NonOverlappingTemplate
 82   5   3   4   0   2   0   1   1   2 0.000000 *   27/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 20   8  14  11  14   3   9   6  10   5 0.006661     96/100     NonOverlappingTemplate
 34  16   5  10   7   9   5   5   6   3 0.000000 *   91/100  *  NonOverlappingTemplate
 64  11   1   5   4   0   4   3   3   5 0.000000 *   51/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 86  10   1   2   0   0   0   0   1   0 0.000000 *   32/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 43  13  12   4  11   4   6   5   1   1 0.000000 *   83/100  *  NonOverlappingTemplate
 52  17   9   7   5   1   2   2   3   2 0.000000 *   85/100  *  NonOverlappingTemplate
 66   3   2   4   6   4   1   5   3   6 0.000000 *   45/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 74   8   2   7   1   0   2   4   1   1 0.000000 *   52/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 80  10   5   0   1   2   1   0   1   0 0.000000 *   41/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 61  11   5   4   8   2   4   0   2   3 0.000000 *   68/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 87   7   3   1   0   1   1   0   0   0 0.000000 *   36/100  *  NonOverlappingTemplate
100   0   0   0   0   0   0   0   0   0 0.000000 *    0/100  *  NonOverlappingTemplate
 56  13   9   7   5   3   1   2   2   2 0.000000 *   81/100  *  NonOverlappingTemplate
```



95	1		0	0	3	0	0	0	0	0.000000 '''	1 . /100	'''	N	nOver lapping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''	N	n er la.ping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''	N	nOver la.ping: em!.)l ate
	9	5	9	9	6	6		3	4	0.000000 '''	80/100	'''	N	nOver la.ping: em!.)l ate
1 0	0		0	0		0	0		0	0. 00 00 '''	/1 0	'''	N	nOver la. ping'Eem plate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''	N	nOver la.ping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 ''	3/100	''	N	nOver la. p.:.ng: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	- 1100	'''	N	nOver la.ping'Eem!.)l ate
.2	10	9	4		3	7	6		5	0.000000 '''	3 /100	'''	N	nOver la.ping: em!.)l ate
-..3	16	10	6	9			8	ti	-5	0. 88 6	9 /100		N	nOver la.ping: em!.)l ate
4	6	6		0		1	0	0	0	0.000000 '''	3_/100	'''	N	nOver lapping: em!.)l ate
38	18		9	8	6			5	4	0.000000	90/1 00	'''	N	nOver la.ping:' em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /1 00	'''	N	nOver la_ping: emplate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /1 00	'''	N	nOver lapping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /1 00	'''	N	n e la_ping: em!.)l ate
.2	2	L	6	6		1		5	1	0.000000 '''	85/100	'''	N	nOver la_ping: em!.)l ate
4	8		6			0	0	0	1	0.000000 '''	4 . /100	'''	N	nOver la_ping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0/ 100	'''	N	nOver la_ping'Eem!_)l ate
97			0	0	0	0	0	0	0	0.000000 '''	- /100	'''	N	nOver la_ping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 ''	0 /100	''	N	nOver la_p.:.ng: em!.)l ate
.6	l'9	13	7	5		3			0	0.000000 '''	86/100	'''	N	nOver la_ping'Eem!.)l ate
6	5		1	0	0	3	1		2	0.000000 '''	- /100	'''	N	nOver la_ping: em!.)l ate
0		5	1	5	3	1	4			0. 00 00 '''	56/1 0	'''	N	nOver la_ping:' emplate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	/100	'''	N	nOver lapping: em!.)l ate
	10			1		1			1	0.000000	46/100	'''	N	nOver la_ping:' em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''	N	nOver la_ping: em!.)l ate
6	1		5	4		6	1		1	0.000000 '''	66/100	'''	N	nOver lapping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''	N	n e la_ping: em!.)l ate
6	13	10	5	9		0		5		0.00009 '''	9 /100	'''	N	nOver la_ping: em!.)l ate
1			.3	1		2	1		1	0 . 00 00 '''	3 /1 0	'''	N	nOver la_ping:' em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0/ 100	'''	N	nOver la_ping'Eem!_)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''		nOver la_ping: em!.)l ate
38	10	L	.3	10	13	7			4	0.000000 '''	89/100	'''		er la_p..:.ng: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''	N	nOver la_ping'Eem!.)l ate
2	6	9	3	1		3	3	0	1	0.000000 '''	55/100	'''	N	nOver la_ping: em!.)l ate
65	11	5	5	1	6	5		0	0	0.000000 '''	45 /100	'''	N	nOver lapp ..:.ng: em!.)l ate
61			6			.3	3	5		0.000000 '''	4 /100	'''	N	nOver lapping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000	0 /100	'''	N	n er la_p .:.ng: em!.)l ate
88			2	1	0	0	0	0	0	0.000000 '''	. /100	'''	N	nOver la_ping: em!.)l ate
.4	l'9		6		3	.3	4		.3	0.000000 '''	86/100	'''	N	nOver lapping: em!.)l ate
100	0	0	0	0	0	0	0	0	0	0.000000 '''	0 /100	'''	N	nOver la_ping: em!.)l ate
0	21	9	9	9	5	2	4			0.000 .14	9 /100		N	nOver la_ping: em!.)l ate

95	2	1	0	2	0	0	0	0	0	0.000000	*	15/100	*	NonOverlappingTemplate
73	8	5	3	2	4	0	1	2	2	0.000000	*	46/100	*	NonOverlappingTemplate
93	3	2	0	1	0	1	0	0	0	0.000000	*	21/100	*	NonOverlappingTemplate
15	12	11	17	12	4	6	8	9	6	0.075719		99/100		NonOverlappingTemplate
95	3	1	0	0	0	0	0	1	0	0.000000	*	7/100	*	NonOverlappingTemplate
84	6	4	1	1	0	3	1	0	0	0.000000	*	24/100	*	NonOverlappingTemplate
100	0	0	0	0	0	0	0	0	0	0.000000	*	3/100	*	NonOverlappingTemplate
100	0	0	0	0	0	0	0	0	0	0.000000	*	4/100	*	NonOverlappingTemplate
74	6	6	5	2	2	2	1	1	1	0.000000	*	35/100	*	NonOverlappingTemplate
88	1	1	4	1	1	1	2	0	1	0.000000	*	27/100	*	OverlappingTemplate
100	0	0	0	0	0	0	0	0	0	0.000000	*	0/100	*	Universal
100	0	0	0	0	0	0	0	0	0	0.000000	*	0/100	*	ApproximateEntropy
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursions
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
0	0	0	0	0	0	0	0	0	0	----		------		RandomExcursionsVariant
100	0	0	0	0	0	0	0	0	0	0.000000	*	0/100	*	Serial
32	11	10	6	13	5	10	2	6	5	0.000000	*	92/100	*	Serial
13	7	16	8	7	10	13	6	10	10	0.419021		99/100		LinearComplexity

- -

The minimum pass rate for each statistical test with the exception of the random excursion (variant) test is approximately = 96 for a sample size = 100 binary sequences.

The minimum pass rate for the random excursion (variant) test is undefined.

For further guidelines construct a probability table using the MAPLE program provided in the addendum section of the documentation.

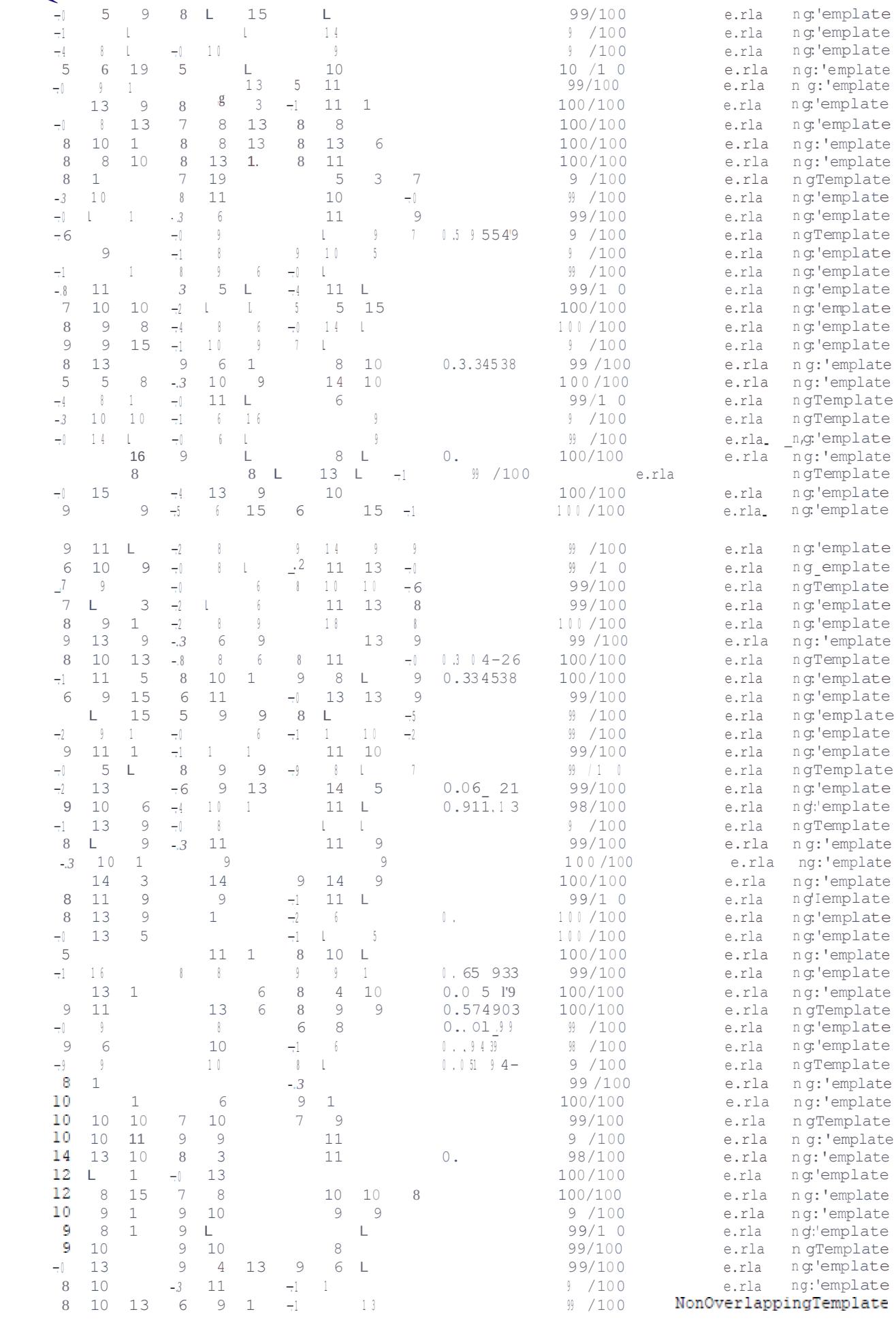
- -

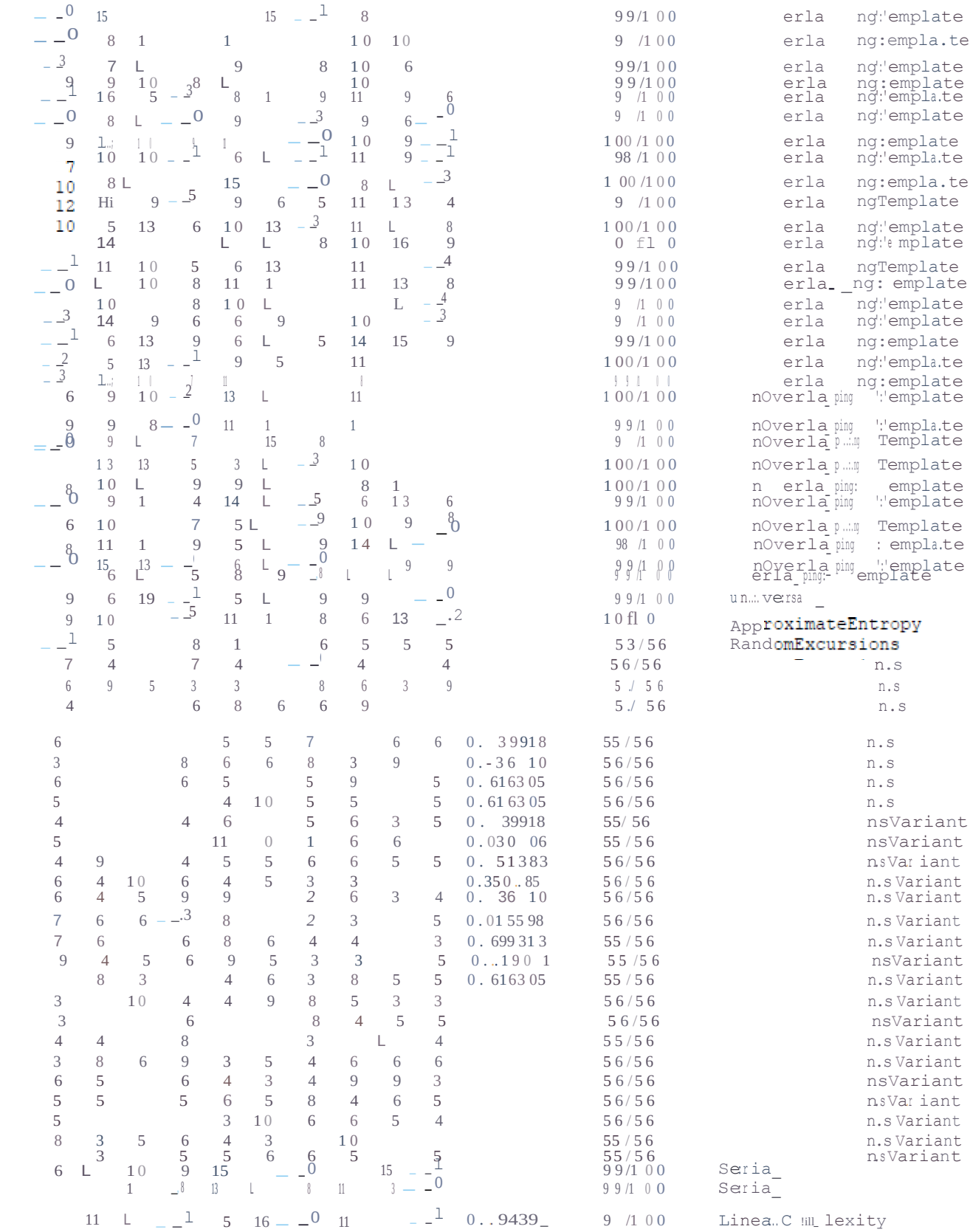
Appendix B: NIST SP 800-22 test results for whitened output

For testing methodology and interpretation of results, see the “Test results” section.

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <lots_white_out_20.0k.bin>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  9   8  11   8  11  13   6   8  14  12 0.739918   100/100     Frequency
 11  14   7   9  13   7  11   9   8  11 0.816537   100/100     BlockFrequency
  7  11  11  10   8  12   5  13   8  15 0.514124    99/100     CumulativeSums
 10   7  12  10  11   6  12  13  11   8 0.851383   100/100     CumulativeSums
 16   7   7  10   7   6  18   7  13   9 0.062821    98/100     Runs
  8  15  14  13   5  11   5   9   8  12 0.249284    99/100     LongestRun
  8   9  12  10  11   9  10  12   9  10 0.996335    99/100     Rank
  9  12   8  10   9  11  11  14   9   7 0.924076    97/100     FFT
  7   7  16   9   7  12  15   8  12   7 0.275709   100/100     NonOverlappingTemplate
  8  15   6  -0   8  13   5  14      -1 0.350.85    99fl 00     N nOverla_ping:'em.)late
  9   1   1   1  14   6   5   9      -0 0.616305   100fl 00     N n  erlapp..g:em.)late
  9  11   L   9  11       5   9      -5 0.595549    99fl 00     N nOverla_ping'Eem.)late
 -0  10  10   9  10      -0  15       8 0.946308     9 /100     N nOverla_ping:'em.)late
     13   1  -3   9       9   9                     9 fl 00         erlapping:'em.)late
 -0  10   L   4  15       8  15                     9 fl 00       n erlapping:'em.)late
 10  13   9  11  11       9   9                    100fl 00       nOverla_pin g:em.)late
     13   6   0   1      -4       6                 9 /100         e.rla_pin g':'em.)late
 -4   6  10  -4   L       4  10  L                  9 /100       n erlapping:'em.)late
  7  15  10   8  11      -0  13  10                 99fl 00     N nOverla_ping:'em.)late
  1   9   9   9          -0   9  L       0.955835   9 /100         e.rla_pin g':'em.)late
     14   1   0   6       9   6  10      0. 9439    9 fl 00       n erlapp..g:em.)late
  4   8   L  12   6   1  -4  16   5      0.090936   100/100     N nOverla_ping'Eem.)late
 -0   8   9  -3   4      -0  11  L      0.534_46   99/100      N nOverla_ping:'em.)late
 -0                 11   1   8   9  16  0.699313   9 fl 00     N n  erlapping:'em.)late
 12   L   6   9   9           L   6                 9 /100       nOverla_ping'Eem.)late
 -1   9      -1   8           10   9                100/100         e.rla_ping: em.)late
 -3   9  10   8   9           10  10                99fl 00         erlapping:'em.)late
  6  11      -1               19   1                100/100      n erlapping:'em.)late
  9  10  15  -3                9                    99fl 00     N nOverla_ping:'em.)late
 10  14       4  14       5   9   1                100/100         e.rla_pin g':'em.)late
 12               -1   8  13   6  10  15           100/100      n erlapping:em.)late
 14               -6   9   L  -3  10               100fl 00     N nOverla_ping:'em.)late
 13   9           8  13   5  -4  11                9 fl  0     N n  e.rla_pin g':'em.)late
  5  16          13  13       5   L                100/100       nOverla_ping'Eemplate
  7   8  10  13  10   9  -1   8  10               99/100        nOverla_ping'Eem.)late
  6   L   L  -5   L   9       16  10               99/100          e.rla_ping : em.)late
  4  10  _o   9  11   5  -3       6                99fl 00         erlapping:'em.)late
 -3       9   9   8       6   9  16               9 fl  0         e.rla_pin g'Eem.)late
  9  10   L   0   8   9  -3       L        63     100/100         e.rla_pin g: emplate
 -0  13  10  -3  15   9   6       6   8  0.53  46   9 fl 00         erlapping:'em.)late
 11       5  10  15   9  -5      10   7  0.38  2    96fl 00       n e.rla_pin g:em.)late
 -0   L  16   6   9   9   9       9   8  0.         99fl 00       nOverla_ping:'em.)late
  6  16              13   L       9   1  _2  0.    10 fl  0      n e.rla_pin g':'em.)late
 11  15  10  10   8          -2       6   8          9 /100     N nOverla_ping'Eemplate
  5  11   9  -4  13          -1           5         100fl 00     N nOverla_ping:'em.)late
  3   9  13   5   L           6                     100fl 00     N n  erlapp..g:em.)late
  9  10  10  10   L          -0                8    99/100        nOverla_ping'Eem.)late
  7               -3  10   1   6   8       _2      10 fl  0         e.rla_pin g'Eem.)late
  2                0  11   5  -4   6   1   -2      99/100           e.rla_pin g: emplate
      9                10   5  -5   9  13   9      9 fl 00     N     erlapping:'em.)late
  8                    9   8  16   8  15   5  -2  99fl 00     N     e.rla_pin g:em.)late
  1  14   9   9   9  13  -1  13   3   8  0..19021 100/100           e.rla_pin g':'em.)late

          1   9  10   9  -0  10      -0  0.63 -19    9 /100     NonOverlappingTemplate
  9   8  13   9  13   5  -3   8  L   -1  0.699313   9 /100     NonOverlappingTemplate
 -0                   9   6  -4      13  -5  0.3 66918  9 /100        e.rla  ngTemplate
  5  11  15  -6  10       -0   9   9      0.3 .345 .38  10 /10        e.rla  ng:'emplate
     13   6   6  13   9   8  L   15   6                9 /100        e.rla  n g:'emplate
  8  13   9  -4   8  15       9       3               100/100        e.rla  ngTemplate
```





The minimum pass rate for each statistical test with the exception of the random excursion (variant) test is approximately = 96 for a sample size = 10 binary sequences.

The minimum pass rate for the random excursion (variant) test is approximately = 53 for a sample size = 56 binary sequences.

For further guidelines construct a probability table using the MATLAB program provided in the addendum section of the documentation.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01 Jan 2024)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.