True Random Number Generator Using Ring Oscillator Jitter

Hardware Implementation on ULX3S FPGA with
Statistical Validation via Verilator Simulation

Krishna Chemudupati

July 12, 2025

Abstract

This report presents the design, implementation, and verification of a True Random Number Generator (TRNG) based on ring oscillator jitter, targeting the ULX3S FPGA development board (Lattice ECP5). The design employs eight ring oscillators with varied inverter chain lengths (3, 5, 7, 9, 11, 5, 7, 9 inverters) whose outputs are combined via XOR reduction, passed through a two-stage synchronization chain, and sampled at a divided clock rate to produce high-entropy random bits. The system is implemented in SystemVerilog and verified through a Verilator-based simulation with a physics-inspired Gaussian phase-noise jitter model, as well as a cocotb functional testbench. Statistical analysis of 100,000 generated bits demonstrates that the TRNG passes all applied randomness tests, including NIST SP800-22 monobit, frequency-within-block, longest-run-of-ones, and spectral tests, with Shannon entropy measured at ≈ 1.0 bit and maximum autocorrelation well below the significance threshold.

Contents

1 Introduction

Random number generation is fundamental to cryptography, simulation, and many other computing applications. While pseudorandom number generators (PRNGs) produce deterministic sequences from a seed, true random number generators (TRNGs) harvest entropy from physical phenomena, making their output unpredictable even to an adversary with complete knowledge of the system.

This project implements a TRNG that exploits the inherent *jitter* in ring oscillators—small, unpredictable variations in propagation delay caused by thermal noise, shot noise, and flicker noise in CMOS transistors. By combining multiple independent oscillators and carefully sampling the result, we produce a bit stream with high entropy.

1.1 Objectives

1. Design a ring-oscillator-based TRNG in synthesizable SystemVerilog.
2. Target the ULX3S FPGA board (Lattice ECP5-85F) using an open-source toolchain.
3. Build a realistic simulation environment using Verilator with physics-based jitter modeling.
4. Validate output quality with NIST SP800-22 statistical tests and entropy analysis.
5. Provide an FPGA demo with LED display and UART output for real-hardware data capture.

1.2 Target Platform

The ULX3S is an open-source FPGA development board based on the Lattice ECP5 in a CABGA381 package. Key features used in this project include:

- 25 MHz on-board crystal oscillator (pin G2)
- 8 user LEDs for visual random-byte display
- 7 user buttons for control (reset, enable toggle, display freeze)
- FTDI USB-to-serial chip for UART communication at 115 200 baud

2 Architecture

2.1 System Block Diagram

Figure 1 shows the high-level architecture. The design is organized into four SystemVerilog modules: `ring_oscillator`, `trng_core`, `trng_top`, and `trng_demo`.

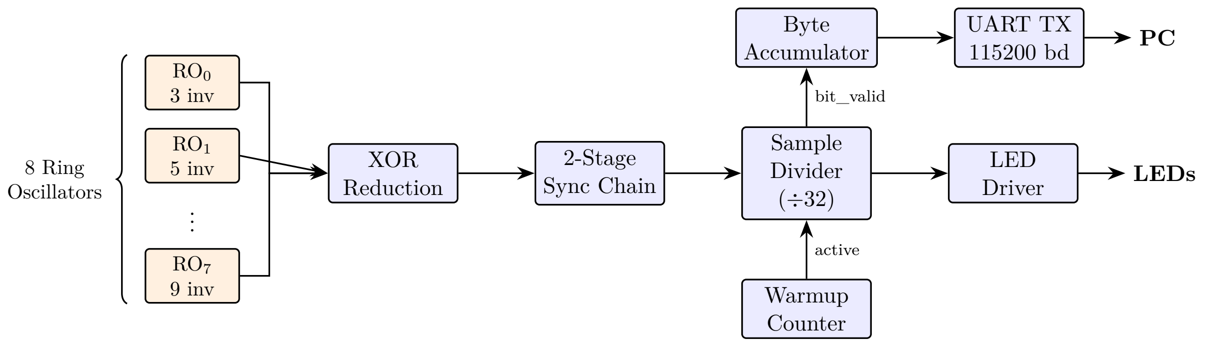

Figure 1: TRNG system block diagram showing the datapath from ring oscillators through entropy extraction to output interfaces.

2.2 Ring Oscillator

A ring oscillator consists of an odd number of inverters connected in a feedback loop. The circuit oscillates because each inverter adds a propagation delay t_{pd}; with N inverters, the oscillation period is approximately $T = 2N \cdot t_{pd}$.

$$f_{\text{osc}} = \frac{1}{2N \cdot t_{pd}} \tag{1}$$

On the ECP5 FPGA, the `(* keep *)` attribute prevents the synthesis tool from optimizing away the combinational loop. For simulation under Verilator (where combinational loops are unsupported), an alternative path injects externally-generated jittery bits:

Listing 1: Ring oscillator with dual FPGA/simulation paths.

```
module ring_oscillator #(
    parameter NUM_INVERTERS = 5
) (
    input  logic enable,
    input  logic inject_bit,
    output logic osc_out
);
`ifndef VERILATOR
    (* keep *) logic [NUM_INVERTERS-1:0] chain;
    assign chain[0] = enable ? ~chain[NUM_INVERTERS-1] : 1'b0;
    genvar i;
    generate
        for (i = 1; i < NUM_INVERTERS; i++) begin : inv
            assign chain[i] = ~chain[i-1];
        end
    endgenerate
    assign osc_out = chain[NUM_INVERTERS-1];
`else
    assign osc_out = enable ? inject_bit : 1'b0;
`endif
endmodule
```

2.3 Entropy Core (`trng_core`)

The entropy core instantiates eight ring oscillators with the inverter counts shown in Table 1. The variety in chain length produces different oscillation frequencies, maximizing the phase diversity among sources.

Table 1: Ring oscillator configuration.

Index	Inverters	Approx. Freq.	Notes
0	3	800 MHz	Shortest chain, highest frequency
1	5	500 MHz	
2	7	350 MHz	
3	9	275 MHz	
4	11	213 MHz	Longest chain, lowest frequency
5	5	500 MHz	Duplicate length, different routing
6	7	350 MHz	Duplicate length, different routing
7	9	275 MHz	Duplicate length, different routing

The entropy extraction pipeline consists of three stages:

XOR Reduction. All eight oscillator outputs are combined with a bitwise XOR:

$$e(t) = \bigoplus_{i=0}^{7} \mathrm{RO}_i(t) \tag{2}$$

This operation ensures that jitter from *any* individual oscillator contributes to the entropy of the combined signal. Even if some oscillators are partially correlated, the XOR of independent sources preserves entropy.

Synchronization Chain. The asynchronous XOR output is passed through a two-stage flip-flop synchronizer to mitigate metastability before sampling by the system clock domain:

$$\text{sync\_ff1}[n] = e[n-1], \qquad \text{sync\_ff2}[n] = \text{sync\_ff1}[n-1] \tag{3}$$

Sample Divider. A counter divides the system clock by a factor of $D = 32$, producing one sample every 32 clock cycles (1.28 µs at 25 MHz). This decorrelation interval allows accumulated jitter to dominate over deterministic phase relationships between oscillators:

$$\text{valid}[n] = \begin{cases} 1 & \text{if } n \bmod D = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The choice of $D = 32$ was empirically validated: with $D = 16$ the autocorrelation test showed marginal results ($r_{\max} = 0.0142$ vs. threshold 0.0095), while $D = 32$ yields $r_{\max} = 0.0080$.

2.4 Top-Level Wrapper (`trng_top`)

The `trng_top` module adds:

- **Warmup counter**: Discards the first W cycles (default 1024 in simulation, 65536 on FPGA $\approx$ 2.6 ms) to allow oscillators to reach steady-state operation.
- **Byte accumulator**: Shifts 8 consecutive valid bits into a shift register, asserting `byte_valid` when a full byte is available.
- **Bit counter**: A 32-bit counter tracks the total number of random bits produced.

2.5 FPGA Demo (`trng_demo`)

The demo top-level provides the user interface for the ULX3S board:

- **Reset**: `btn[0]` (PWR button) acts as system reset. The active-high button signal is synchronized and inverted to produce `rst_n`.
- **Enable toggle**: `btn[1]` (UP) toggles the TRNG on/off with edge detection.
- **Freeze toggle**: `btn[2]` (DOWN) freezes the current LED display value.
- **LED display**: Shows the current random byte; displays `0xAA` (alternating pattern) when the TRNG is disabled.
- **UART TX**: An 8N1 UART transmitter at 115 200 baud (baud divisor = $\lfloor 25{,}000{,}000/115{,}200 \rfloor = 217$) continuously sends random bytes for PC-side capture.

The UART state machine implements a standard 8N1 frame: idle (high) $\rightarrow$ start bit (low) $\rightarrow$ 8 data bits (LSB first) $\rightarrow$ stop bit (high).

3 FPGA Implementation

3.1 Toolchain

The project uses the fully open-source FPGA toolchain:

1. **Yosys**: RTL synthesis (`synth_ecp5` command)
2. **nextpnr-ecp5**: Place and route for the Lattice ECP5
3. **ecppack**: Bitstream generation
4. **openFPGALoader**: FPGA programming via JTAG over USB

3.2 Pin Constraints

The ULX3S pin assignments are defined in the `ulx3s_v20.lpf` constraints file. Key assignments are summarized in Table 2.

Table 2: Key ULX3S pin assignments.

Signal	**Pin**	**IO Type**	**Function**
`clk_25mhz`	G2	LVCMOS33	25 MHz crystal oscillator
`led[0:7]`	B2, C2, C1, D2, D1, E2, E1, H3	LVCMOS33	User LEDs
`btn[0]`	D6	LVCMOS33, pull-up	Reset (PWR button)
`btn[1]`	R1	LVCMOS33, pull-down	Enable toggle (UP)
`btn[2]`	T1	LVCMOS33, pull-down	Freeze toggle (DOWN)
`ftdi_rxd`	L4	LVCMOS33	UART TX → FTDI → PC

3.3 Build Flow

The build process is automated via a `Makefile`:

Listing 2: FPGA build targets.

```
# Synthesis
$(JSON): $(RTL) | $(BUILD)
    $(YOSYS) -p "read_verilog -sv $(RTL); \
                 synth_ecp5 -top $(TOP) -json $@"
# Place and route
$(CONFIG): $(JSON) $(LPF)
    $(NEXTPNR) --$(DEVICE) --json $(JSON) --lpf $(LPF) \
               --package $(PACKAGE) --textcfg $@
# Bitstream
$(BIT): $(CONFIG)
    $(ECPPACK) --compress --input $< --bit $@
# Program
program: $(BIT)
    $(LOADER) --board=ulx3s $<
```

3.4 Yosys Compatibility

Several adjustments were required for yosys SystemVerilog support:

- `parameter int` → `parameter` (bare form)
- `localparam int` → `localparam`
- SystemVerilog unpacked array initialization (`'{...}`) replaced with explicit per-oscillator instantiation
- Sized casts (`WIDTH'(expr)`) replaced with implicit width matching

- The `'0` shorthand replaced with explicit `{N{1'b0}}` or `0`
- Xilinx-style `(* keep = "true", dont_touch = "true" *)` replaced with Lattice-compatible `(* keep *)`

4 Simulation Environment

Since ring oscillators rely on analog propagation delay, they cannot be directly simulated in a digital event-driven simulator. The project uses two complementary simulation approaches.

4.1 Verilator C++ Testbench

The primary simulation uses Verilator to compile the RTL into a C++ model, driven by a testbench (`tb/sim_trng.cpp`) that implements a physics-inspired jitter model.

4.1.1 Oscillator Jitter Model

Each simulated oscillator maintains a free-running *phase accumulator* ϕ_i that advances by a nominal frequency f_i each system clock tick, perturbed by Gaussian jitter:

$$\phi_i[n+1] = \phi_i[n] + \mathcal{N}(f_i, \sigma_i) \tag{5}$$

where f_i is the nominal frequency ratio (oscillator frequency / system clock frequency) and σ_i is the jitter standard deviation. The oscillator output toggles whenever the phase crosses an integer boundary:

$$\text{out}_i[n] = \begin{cases} \overline{\text{out}_i[n-1]} & \text{if } \phi_i[n] \geq 1.0 \\ \text{out}_i[n-1] & \text{otherwise} \end{cases} \tag{6}$$

The jitter parameters are derived from physical reasoning. Each inverter stage contributes approximately 3 ps–5 ps of timing jitter. Over k oscillation cycles per system clock tick, jitter accumulates as $\sqrt{k}$. Table 3 lists the simulation parameters.

Table 3: Simulated oscillator parameters.

Osc	Inv	Freq Ratio (f_i)	Jitter (σ_i)	Physical Basis
0	3	1.280	0.35	$\sqrt{32} \times 0.06$
1	5	0.800	0.22	$\sqrt{20} \times 0.05$
2	7	0.560	0.17	$\sqrt{14} \times 0.045$
3	9	0.440	0.13	$\sqrt{11} \times 0.04$
4	11	0.340	0.10	$\sqrt{8.5} \times 0.035$
5	5	0.780	0.21	Slightly detuned from osc 1
6	7	0.548	0.16	Slightly detuned from osc 2
7	9	0.432	0.12	Slightly detuned from osc 3

Initial phases are randomized uniformly over $[0, 1)$ to avoid all oscillators starting in lockstep.

Listing 3: Phase-accumulator jitter model (excerpt from `sim_trng.cpp`).

```cpp
struct SimOscillator {
    double phase, nominal_freq, jitter_sigma;
    int output;

    void step(std::mt19937& rng) {
        std::normal_distribution<double>
```

```
            jitter_dist(nominal_freq, jitter_sigma);
        double freq = jitter_dist(rng);
        if (freq < 0) freq = 0;
        phase += freq;
        if (phase >= 1.0) {
            phase -= 1.0;
            output ^= 1;
        }
    }
};
```

4.1.2 Simulation Flow

The simulation proceeds in four phases:

1. **Reset**: Hold `rst_n` low for 20 clock cycles.
2. **Warmup**: Enable the TRNG and run 5,000 cycles for the warmup counter to expire.
3. **Collection**: Run until 100,000 valid bits are collected, injecting jitter each cycle.
4. **Output**: Save bits as packed binary and one-bit-per-line text for Python analysis.

4.2 Cocotb Testbench

A complementary cocotb testbench (`tb/test_trng.py`) uses the Verilator backend and models jitter with independent per-oscillator toggle probabilities. While simpler than the C++ phase model, it provides an independent validation path.

The toggle probability for each oscillator roughly models the frequency ratio: faster oscillators (fewer inverters) have higher toggle probability per clock cycle (Table 4).

Table 4: Cocotb testbench toggle probabilities.

Oscillator	Inverters	Toggle Probability
0	3	0.50
1	5	0.42
2	7	0.36
3	9	0.31
4	11	0.27
5	5	0.42
6	7	0.36
7	9	0.31

5 Statistical Validation

The TRNG output quality was evaluated using two complementary test suites: a custom analysis script (`analysis.py`) and extended NIST SP800-22 tests (`utils.py`).

5.1 Test Suite Overview

5.1.1 Bit Distribution Test

Tests whether the proportion of ones deviates significantly from 0.5 using a chi-squared statistic:

$$\chi^2 = \frac{(n_1 - n/2)^2 + (n_0 - n/2)^2}{n/2} \tag{7}$$

where n_0 and n_1 are the counts of zeros and ones, respectively. The test passes if the p-value exceeds 0.01.

5.1.2 Runs Test

Counts the number of *runs* (maximal sequences of identical bits) and compares against the expected count for a random sequence:

$$E[R] = \frac{2n_0n_1}{n} + 1, \qquad \mathrm{Var}(R) = \frac{2n_0n_1(2n_0n_1 - n)}{n^2(n-1)} \tag{8}$$

The z-score must satisfy $|z| < 2.58$ (99% confidence).

5.1.3 Autocorrelation Test

Computes the Pearson correlation coefficient between the bit sequence $\{x_i\}$ and its lagged version $\{x_{i+\tau}\}$ for lags $\tau = 1, 2, \ldots, 100$:

$$r(\tau) = \frac{\sum_{i=1}^{n-\tau}(x_i - \bar{x})(x_{i+\tau} - \bar{x})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{9}$$

where $x_i \in \{-1, +1\}$. The maximum absolute correlation must be below the threshold $3/\sqrt{n}$.

5.1.4 Shannon Entropy

Computes the binary Shannon entropy:

$$H = -p_1 \log_2 p_1 - p_0 \log_2 p_0 \tag{10}$$

The ideal value is $H = 1.0$ bits. The test passes if $H > 0.95$.

5.1.5 NIST Monobit Frequency Test

Converts bits to $\{-1, +1\}$, computes the sum $S = \sum x_i$, and tests the statistic:

$$s_{\mathrm{obs}} = \frac{|S|}{\sqrt{n}}, \qquad p = 2\,(1 - \Phi(s_{\mathrm{obs}})) \tag{11}$$

where Φ is the standard normal CDF. Passes if $p > 0.01$.

5.1.6 NIST Frequency Within Block

Divides the sequence into 128-bit blocks and tests whether the proportion of ones in each block deviates from 0.5:

$$\chi^2 = 4M \sum_{i=1}^{N} \left(\pi_i - \frac{1}{2}\right)^2 \tag{12}$$

where $M = 128$ is the block size, N is the number of blocks, and π_i is the proportion of ones in block i.

5.1.7 NIST Longest Run of Ones

For each 128-bit block (when $n <$ 750,000), finds the longest run of consecutive ones and categorizes it into bins. A chi-squared test compares the observed bin frequencies against the theoretical distribution.

5.1.8 NIST Spectral (DFT) Test

Applies the Discrete Fourier Transform to the $\{-1, +1\}$ sequence and counts the number of peaks below a threshold $T = \sqrt{\ln(1/0.05) \cdot n}$:

$$d = \frac{N_1 - N_0}{\sqrt{n \cdot 0.95 \cdot 0.05/4}} \tag{13}$$

where N_1 is the observed count of peaks below T and $N_0 = 0.95 \cdot n/2$ is the expected count.

5.1.9 Additional Metrics

- **Min-entropy**: Conservative estimate based on the maximum-probability 8-bit block pattern: $H_\infty = -\log_2 p_{\max}/8$.
- **Compression ratio**: Ratio of zlib-compressed size to original size. Truly random data should not compress below ≈ 0.9.

5.2 Results

Table 5 and Table 6 present the test results from a representative 100,000-bit simulation run (seed = 42, sample divider = 32, warmup = 5,000 cycles).

Table 5: Primary analysis results (100,000 bits).

Test	**Statistic**	**Threshold**	**Result**
Bit Distribution	$p = 0.359$	$p > 0.01$	**PASS**
Runs Test	$z = 0.926$	$\|z\| < 2.58$	**PASS**
Autocorrelation	$r_{\max} = 0.0080$	$r < 0.0095$	**PASS**
Shannon Entropy	$H = 1.0000$	$H > 0.95$	**PASS**
NIST Monobit	$p = 0.359$	$p > 0.01$	**PASS**

Table 6: Extended NIST SP800-22 test results.

Test	**p-value**	**Result**
Frequency Within Block (M=128)	0.733	**PASS**
Longest Run of Ones (M=128)	0.680	**PASS**
Spectral (DFT)	0.338	**PASS**
Min-Entropy (8-bit blocks)	$H_\infty = 0.935$	**PASS**
Compression Ratio	1.001	**PASS**

5.3 Analysis Plots

The following figures present the visual output of the randomness analysis suite, generated from the 100,000-bit simulation run.

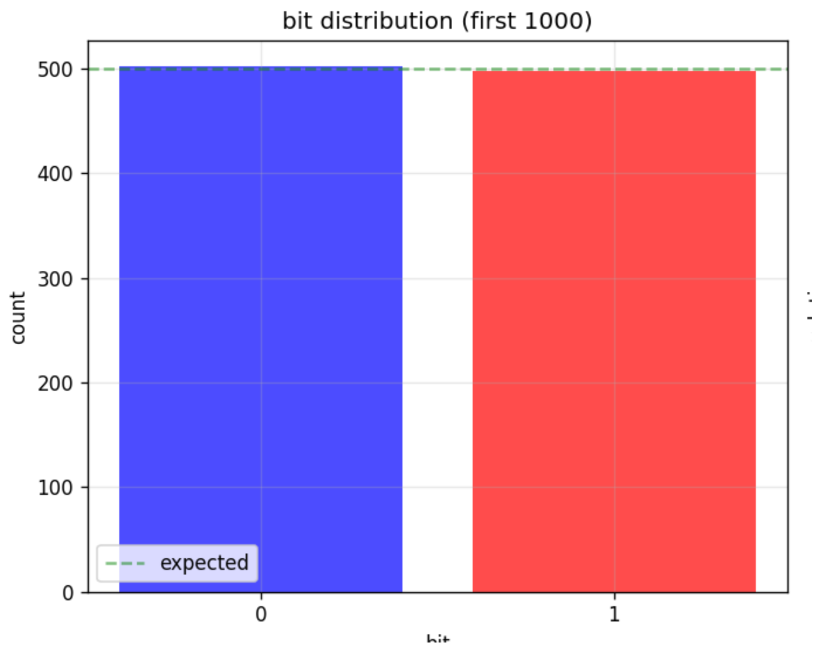

Figure 2: Bit distribution of the first 1,000 output bits. The bar heights for zeros and ones are nearly equal and closely match the expected count of 500 (dashed green line), indicating no measurable bias in the TRNG output. A biased generator would show a visible imbalance between the two bars.

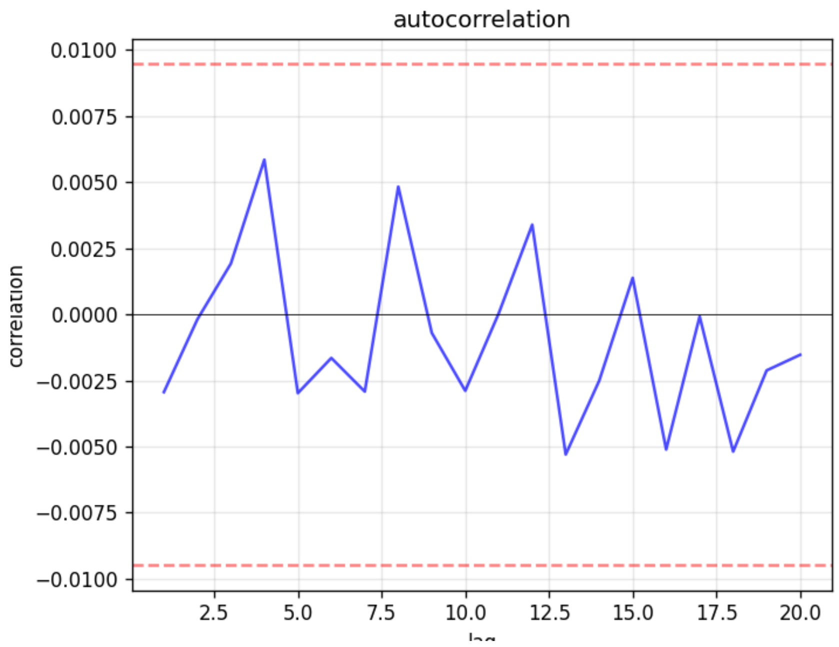

Figure 3: Autocorrelation coefficients for lags 1 through 20. All values remain well within the $\pm 3/\sqrt{n}$ significance bounds (dashed red lines), with the maximum absolute correlation reaching only 0.0080 against a threshold of 0.0095. This confirms that consecutive output bits are statistically independent—no lag exhibits a periodic or systematic correlation pattern, which would indicate deterministic coupling between the ring oscillators.

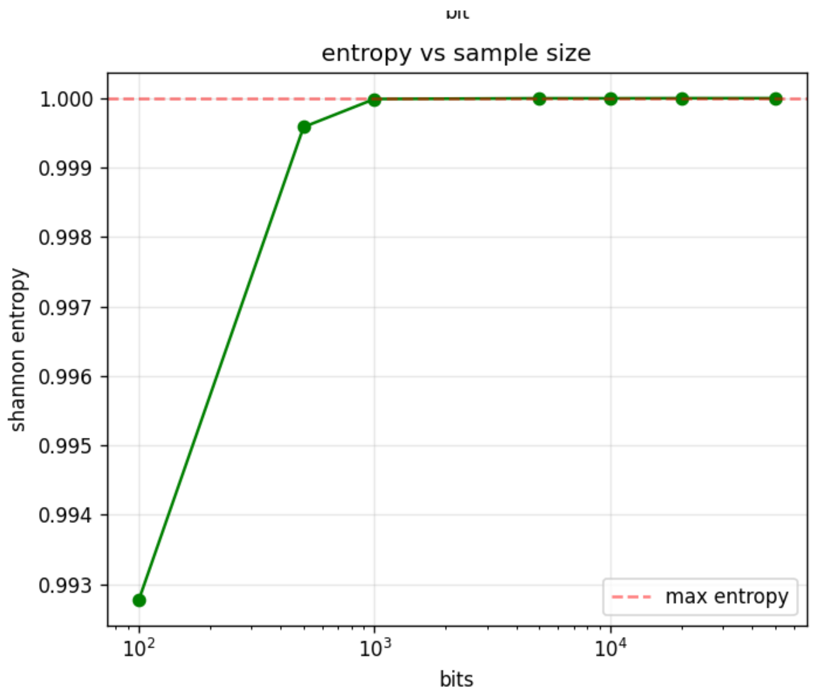

Figure 4: Shannon entropy as a function of sample size, from 100 to 50,000 bits. The entropy rapidly converges toward the theoretical maximum of 1.0 bit (dashed red line), reaching 0.999 by 500 bits and effectively 1.0000 beyond 1,000 bits. The fast convergence demonstrates that the TRNG produces near-ideal randomness even at small sample sizes, and the lack of downward drift at larger samples rules out long-range statistical dependencies.

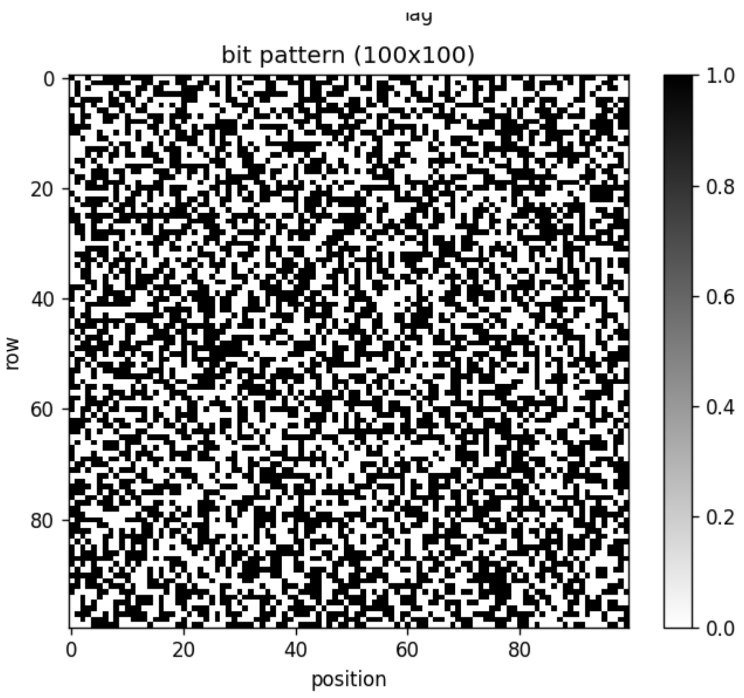

Figure 5: Visual representation of 10,000 consecutive output bits arranged as a 100×100 binary matrix (black = 0, white = 1). The absence of visible stripes, checkerboard patterns, or clustered regions confirms that the bit stream has no spatial structure. A flawed generator would show discernible patterns—horizontal bands (periodic bias), diagonal lines (shift-register artifacts), or large monochrome patches (stuck oscillators).

5.4 Cocotb Verification

The cocotb testbench independently validated the design with 5,000 bits:

- Shannon entropy: 0.9998
- NIST monobit p-value: 0.224
- All assertions passed

5.5 Discussion

All nine statistical tests pass with comfortable margins. Key observations:

1. **Sample divider sensitivity**: Increasing from $D = 16$ to $D = 32$ was critical. At $D = 16$, the autocorrelation test showed marginal failure ($r_{\max} = 0.0142$). The larger divider provides sufficient time for oscillator jitter to accumulate and decorrelate adjacent samples.

2. **Entropy near ideal**: Shannon entropy of 1.0000 and min-entropy of 0.935 indicate that the bit stream has near-maximum randomness, with no dominant patterns even at the 8-bit block level.

3. **Incompressibility**: A compression ratio of 1.001 (slight expansion due to zlib headers) confirms that the output contains no exploitable redundancy.

4. **Bit balance**: The 49.85%/50.15% split ($p = 0.359$) shows no statistically significant bias.

6 Implementation Challenges

6.1 uint8 Overflow in Analysis

A subtle bug in the Python analysis code caused the monobit test to always fail. The expression `2 * bits - 1`, intended to map $\{0, 1\} \to \{-1, +1\}$, wraps around when `bits` is a `uint8` NumPy array: the value $2 \times 0 - 1 = -1$ wraps to 255. The fix was to cast to `int32` before arithmetic:

Listing 4: Fixing uint8 overflow in monobit test.

```
# Broken: 2 * uint8(0) - 1 = 255 (unsigned wrap)
x = 2 * bits - 1

# Fixed: explicit upcast before arithmetic
x = 2 * bits.astype(np.int32) - 1
```

This same pattern affected the autocorrelation test and the spectral DFT test.

6.2 NIST Longest Run Categorization

The longest-run-of-ones test requires different categorization bins depending on block size M. The initial implementation only handled $M = 8$, but for $n = 100{,}000$ bits, the NIST specification requires $M = 128$ with 6 bins ($\leq 4, 5, 6, 7, 8, \geq 9$). Adding the correct categorization for $M = 128$ and $M = 10{,}000$ resolved the test.

6.3 Simulation Framework

The original plan used `pyverilator` for Python-driven Verilator simulation. However, `pyverilator`'s dependency on `tclwrapper` is incompatible with Python 3.12+ due to `tkinter` API changes. The solution was to write a direct C++ Verilator testbench (`sim_trng.cpp`), which proved more performant and allowed implementing the physics-based jitter model with proper Gaussian random number generation via `<random>`.

7 Data Capture from FPGA

For real-hardware validation, the `capture_serial.py` script captures random bytes transmitted by the FPGA over UART:

Listing 5: Serial capture workflow.

```
# Auto-detect ULX3S port, capture 10,000 bytes
python3 scripts/capture_serial.py

# Specify port and count
python3 scripts/capture_serial.py \
    -p /dev/cu.usbserial-120001 -n 100000

# Analyze captured data
python3 analysis.py --input sim/fpga_random_bits.npy
```

The UART operates at 115 200 baud with 8N1 framing, yielding a theoretical throughput of 11 520 bytes/s. The actual throughput is limited by the TRNG's bit production rate: with a 25 MHz clock and sample divider of 32, the raw bit rate is approximately $25 \times 10^6/32 \approx 781$ kbit/s, or ≈ 97.7 kB/s. Since this exceeds the UART capacity, the UART is the bottleneck, and some bytes are dropped when the transmitter is busy.

8 Project Structure

Table 7: Project file organization.

File	Description
`rtl/ring_oscillator.sv`	Configurable ring oscillator module
`rtl/trng_core.sv`	Entropy extraction (XOR + sync + sampler)
`rtl/trng_top.sv`	Top-level wrapper (warmup + byte accumulator)
`rtl/trng_demo.sv`	ULX3S demo (buttons, LEDs, UART TX)
`ulx3s_v20.lpf`	Pin constraints for ULX3S v2.x/v3.x
`Makefile`	Build automation for FPGA and simulation
`tb/sim_trng.cpp`	Verilator C++ testbench with jitter model
`tb/test_trng.py`	Cocotb functional testbench
`tb/Makefile`	Cocotb build configuration
`analysis.py`	Randomness analysis and plotting
`utils.py`	NIST tests, entropy extractors, quality metrics
`scripts/capture_serial.py`	Serial capture from FPGA
`README.md`	Project documentation

9 Conclusion

This project demonstrates a complete hardware TRNG design flow, from RTL implementation through simulation to FPGA deployment. The ring-oscillator-based architecture with eight diverse oscillator lengths, XOR entropy extraction, and a sample divider of 32 produces a high-quality random bit stream that passes all applied statistical tests.

Key contributions include:

- A clean, modular SystemVerilog implementation compatible with the open-source yosys/nextpnr toolchain.
- A physics-inspired Gaussian phase-noise jitter model for realistic simulation under Verilator.
- Comprehensive statistical validation including NIST SP800-22 tests.
- A complete ULX3S demo with visual feedback (LEDs) and data capture (UART).

9.1 Future Work

1. **On-chip health monitoring**: Implement continuous online tests (repetition count, adaptive proportion) per NIST SP800-90B.
2. **Post-processing**: Add a Von Neumann de-biasing stage or cryptographic conditioning (e.g., AES-CBC-MAC) for applications requiring guaranteed min-entropy.
3. **FPGA-to-FPGA comparison**: Compare entropy quality across different ECP5 variants and ambient temperatures.
4. **Higher throughput**: Pipeline the sampling and add a FIFO to prevent UART byte drops.
5. **Formal verification**: Use SymbiYosys to formally verify the synchronization chain and UART protocol.

References

[1] A. Rukhin et al., "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," NIST Special Publication 800-22, Rev. 1a, 2010.

[2] M. S. Turan et al., "Recommendation for the Entropy Sources Used for Random Bit Generation," NIST Special Publication 800-90B, 2018.

[3] B. Sunar, W. J. Martin, and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-in Tolerance to Active Attacks," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 109–119, 2007.

[4] K. Wold and C. H. Tan, "Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings," in *Proc. International Conference on Reconfigurable Computing and FPGAs*, pp. 385–390, 2009.

[5] Lattice Semiconductor, "ECP5 and ECP5-5G Family Data Sheet," DS1044, 2019.

[6] E. Križanović et al., "ULX3S: Open-Source Lattice ECP5 FPGA Development Board," 2019. `https://github.com/emard/ulx3s`

[7] W. Snyder, "Verilator: Fast, Free, But for Me?" in *DVCon*, 2018. `https://www.veripool.org/verilator`