COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Department of Electronics

Internship Project Report

on

ASIC Design and Implementation of Adders and TRNG using Cadence Tools

Under the Guidance of

Dr. Tripti S Warrier, **Ms. Simi Sukumaran**, **Mr. Adarsh K**

Submitted by

Sreenesh K.S

u2201199@rajagiri.edu.in

Rajagiri School of Engineering & Technology (RSET)

Cochin, Kerala

June 4, 2025

Abstract

This report details the design and ASIC implementation of two digital systems: a True Random Number Generator (TRNG) . The TRNG is based on multiple ring oscillators that generate entropy from physical jitter, with Von Neumann debiasing applied to improve output randomness. It produces 32-bit random numbers suitable for cryptographic applications. Synthesis was carried out using Cadence tools to evaluate key ASIC design metrics such as timing, area, power consumption, and gate count, and simulation was also performed to verify functional correctness.

Contents

1 Introduction

This project involves the ASIC design and implementation of a True Random Number Generator (TRNG) The TRNG uses ring oscillators and Von Neumann debiasing to generate unbiased random bits suitable for cryptographic use.

2 Design and Implementation Tools

- **Design Language:** Verilog HDL
- **Synthesis Tool:** Cadence Genus
- **Simulation and Verification:** Cadence Xcelium and Cadence IMC
- **Analysis Metrics:** Timing, Area, Power Consumption, and Gate Count

3 System Architecture

3.1 TRNG Design

The True Random Number Generator (TRNG) is based on multiple ring oscillators whose outputs are sampled to extract entropy caused by jitter. The sampled outputs are passed through a XOR tree and then processed by a Von Neumann debiasing module to remove bias and improve randomness quality.

- **Ring Oscillators:** Generate jitter-based signals.
- **Sampling Logic:** Flip-flops sample the outputs at a fixed frequency.
- **XOR Tree:** Combines oscillator outputs into a single entropy bit.
- **Von Neumann Debiasing:** Pairs of bits are processed to eliminate bias.

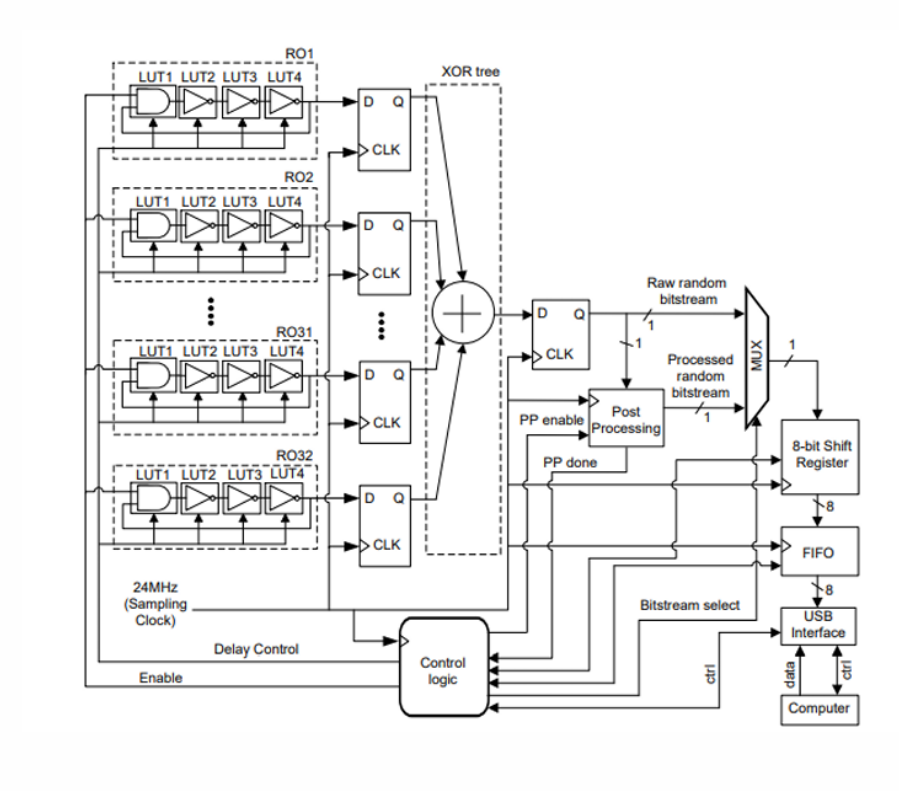


Figure 1: Block Diagram of TRNG Architecture

Each architecture was designed using Verilog HDL and synthesized using Cadence tools to compare performance metrics such as timing, power, and area.

4 Synthesis and Simulation Results

4.1 True Random Number Generator (TRNG)

4.1.1 Simulation Waveform

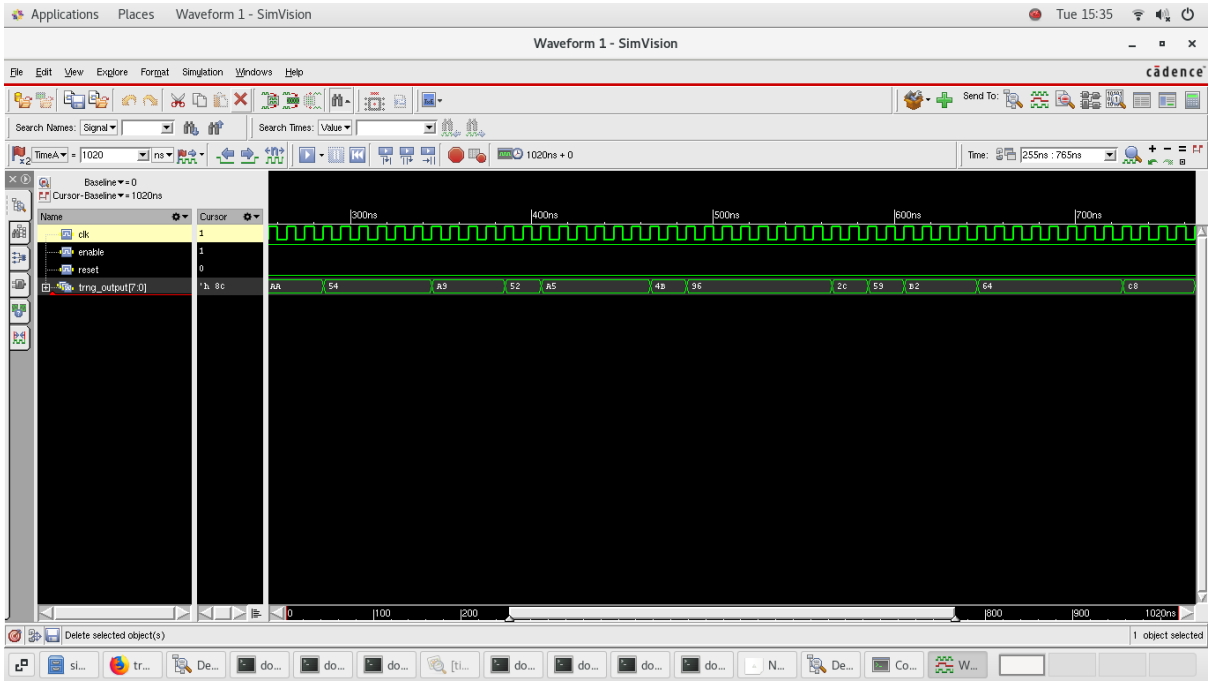


Figure 2: TRNG Waveform Simulation Report Screenshot

4.1.2 Timing Report of TRNG

The timing report shown corresponds to a setup timing analysis for the TRNG design. The key points from the report are:

- **Clock Edges:** Launch at 0 ps and Capture at 10,000 ps (i.e., 10 ns or 100 MHz clock).
- **Setup Time:** 136 ps.
- **Clock Uncertainty:** 50 ps.
- **Required Time:** 9814 ps = 10,000 ps - (136 ps + 50 ps).
- **Data Arrival Time:** 2219 ps.
- **Slack:** 7595 ps = 9814 ps - 2219 ps.
- **Timing Path:** The path starts from a flip-flop (`DFFRX1`) and passes through several logic gates like `AND2X1`, `NAND3X1`, `NOR4BBX1`, etc., finally arriving at another flip-flop (`DFFRHQX1`).

Timing Summary Table

Metric	Value	Description
Clock Period	10,000 ps	Clock cycle time (10 ns for 100 MHz)
Required Time	9814 ps	Time by which data must arrive
Data Arrival Time	2219 ps	Actual arrival time of data at endpoint
Slack	7595 ps	Positive slack (timing met successfully)
Setup + Uncertainty	186 ps	Total setup margin (136 ps + 50 ps)
Data Path Delay	2219 ps	Delay from source to destination
Critical Path	Flip-flop to Flip-flop via logic	E.g., `DFFRX1` → `AND2X1` → ... → `DFFRHQX1`

Table 1: Summary of Setup Timing Analysis

Conclusion of Timing Report

The design meets all setup timing requirements with a large positive slack of 7595 ps. This indicates the circuit is operating well within its timing constraints and is suitable for a 100 MHz clock frequency.

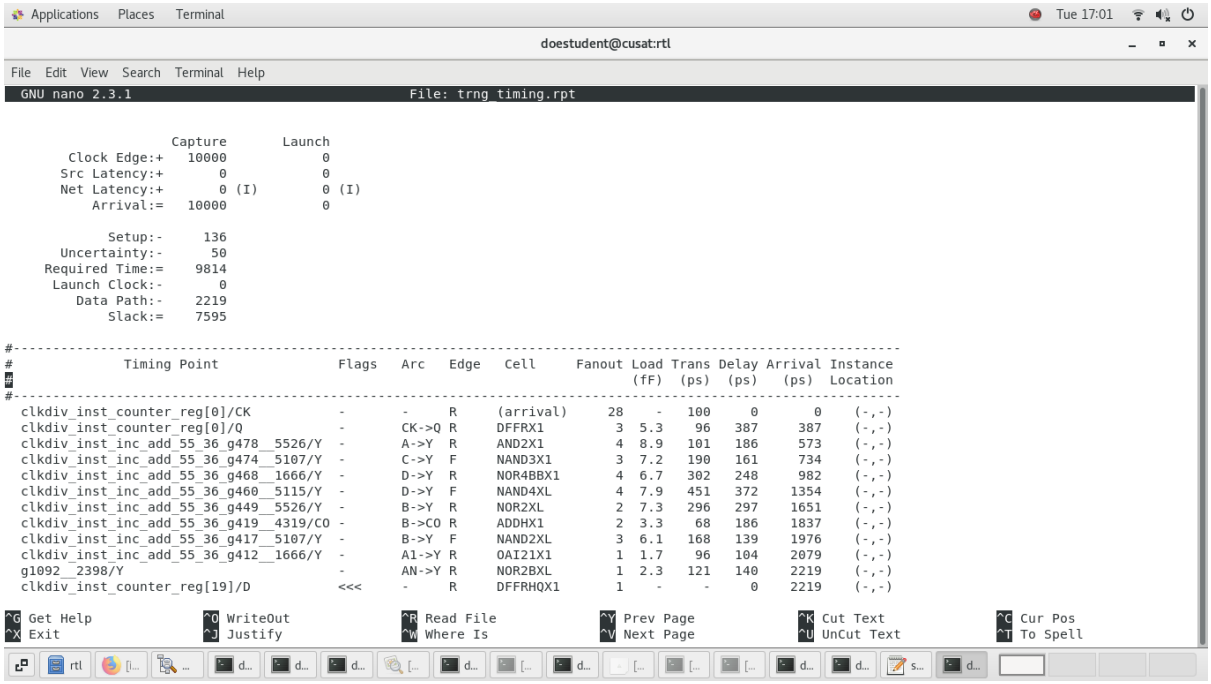


Figure 3: TRNG Timing Report Screenshot

4.1.3 Area Utilization

The table below summarizes the area details:

Table 2: Area Report for `trng_fpga`

Instance	Module	Cell Count	Cell Area	Net Area	Total Area	Wireload
trng_fpga	trng_fpga	138	1299.597	0.000	1299.597	<none>(D)

Notes:

- **Cell Count:** The number of standard cells used is 138.
- **Cell Area:** The total area consumed by these cells is 1299.597 units.
- **Net Area:** Net wiring area is 0.000, indicating negligible or unaccounted routing.
- **Total Area:** Equal to cell area since net area is zero.
- **Wireload:** Default wireload model from the technology library was used.

```
=============================================================
  Generated by:           Genus(TM) Synthesis Solution 20.11-s111_1
  Generated on:           Jun 03 2025  04:58:13 pm
  Module:                 trng_fpga
  Technology library:     slow
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
=============================================================

 Instance Module  Cell Count  Cell Area  Net Area   Total Area   Wireload
-------------------------------------------------------------------------------
trng_fpga                138   1299.597     0.000     1299.597 <none> (D)
  (D) = wireload is default in technology library
```

Figure 4: TRNG Area Report Screenshot

4.1.4 Power Utilization

The table below summarizes the power details:

Table 3: Power Consumption (W)

Component	Total Power (W)	Share (%)
Registers	6.15e-05	87.10%
Logic	5.36e-06	7.59%
Clock	3.75e-06	5.31%

Table 4: Power Consumption in detail (W)

Category	Total Power (W)	Share (%)
Leakage	6.057e-05	8.6%
Internal	5.86e-06	82.96%
Switching	5.976e-06	8.46%
Total Power	**7.06e-05**	**100%**

Inference: The analysis reveals that registers dominate the dynamic power consumption, accounting for 87.1% of the total power, which indicates the design is heavily register-intensive. In contrast, logic and clock components consume significantly less power, contributing only about 7.6% and 5.3%, respectively. This suggests that optimizing register usage could provide the most effective power reduction in this design.

- **Internal power** dominates at **82.96%**, showing that dynamic switching inside the cells is the major contributor.
- **Leakage power** is relatively low (8.58%), indicating efficient transistor usage.
- **Clock network** only consumes switching power, and contributes 3.75e-06 W, which is notable.

```
Instance: /trng_fpga
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
       memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     register     4.33099e-06  5.58855e-05  1.27933e-06  6.14958e-05  87.10%
        latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        logic     1.72626e-06  2.68716e-06  9.42671e-07  5.35610e-06   7.59%
         bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        clock     0.00000e+00  0.00000e+00  3.75030e-06  3.75030e-06   5.31%
          pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
           pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
    Subtotal     6.05725e-06  5.85727e-05  5.97230e-06  7.06022e-05 100.00%
  Percentage           8.58%       82.96%        8.46%      100.00% 100.00%
  ------------------------------------------------------------------------
```

Figure 5: TRNG Power Report Screenshot

4.1.5 Gate Count & Utilization

The table below summarizes the Gate utilization details:

Table 5: Area by Functional Type

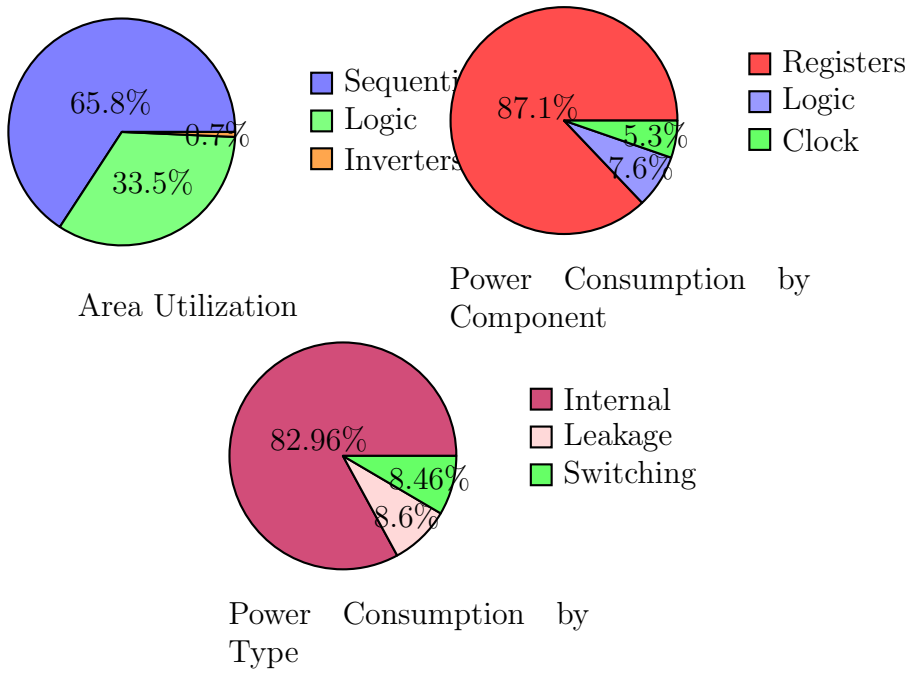
Type	**Area**	**Area %**
Sequential	854.540	65.8%
Logic	435.974	33.5%
Inverters	9.083	0.7%

Inference:

- Sequential logic dominates with 65.8% area, supporting high dependence on registers and flip-flops.
- Inverter count and area are minimal, showing low standalone inversion use.
- D flip-flops (like DFFRHQX1) account for over 50% of total area, emphasizing the design's sequential nature.
- Complex combinational gates such as AO22X1 appear but have lesser area usage.

Type	Instances	Area	Area %
sequential	41	854.540	65.8
inverter	4	9.083	0.7
logic	93	435.974	33.5
physical_cells	0	0.000	0.0
total	138	1299.597	100.0

Figure 6: TRNG Gate Utilization Screenshot



4.1.6 Verification Coverage Inference

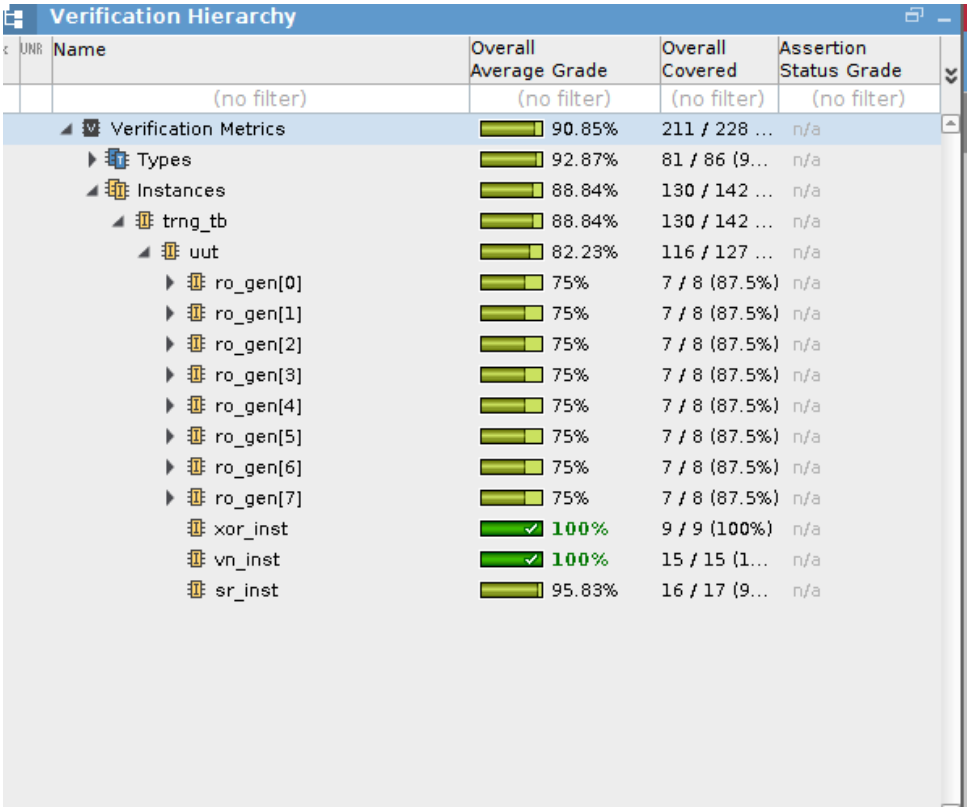


Figure 7: Coverage

The functional verification of the `trng_fpga` design was evaluated using simulation-based metrics. The overall average coverage achieved was **90.85%**, indicating a high level of confidence in the functional correctness of the design.

- The ring oscillator generator modules (`ro_gen[0]` to `ro_gen[7]`) each achieved **75%** coverage, suggesting consistent but partial stimulation. Additional test scenarios may be required to enhance their coverage.
- The critical logic blocks such as the XOR module (`xor_inst`), validation logic (`vn_inst`), and shift register (`sr_inst`) achieved **100%**, **93.3%**, and **95.8%** coverage respectively. This confirms comprehensive validation of the key entropy processing and output logic.
- The testbench instance `trng_tb` achieved **88.84%** coverage, verifying that the testbench design is effective but could be refined to cover more edge cases.

Inference: The high overall coverage and complete verification of core logic blocks confirm the robustness of the design. The ring oscillator coverage can be improved with directed or constrained-random tests to maximize verification completeness.

4.1.7 Schematic Diagrams of TRNG Circuit

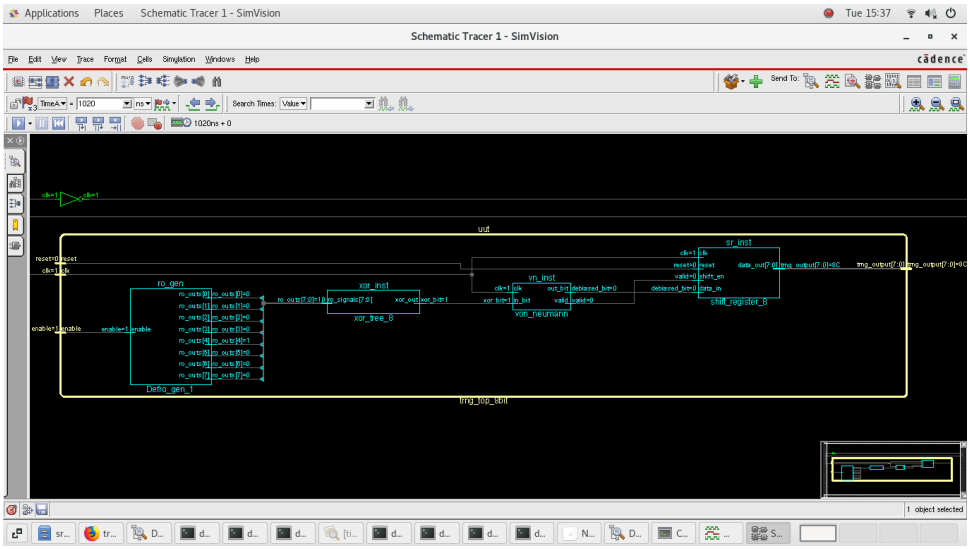


Figure 8: Overall Schematic

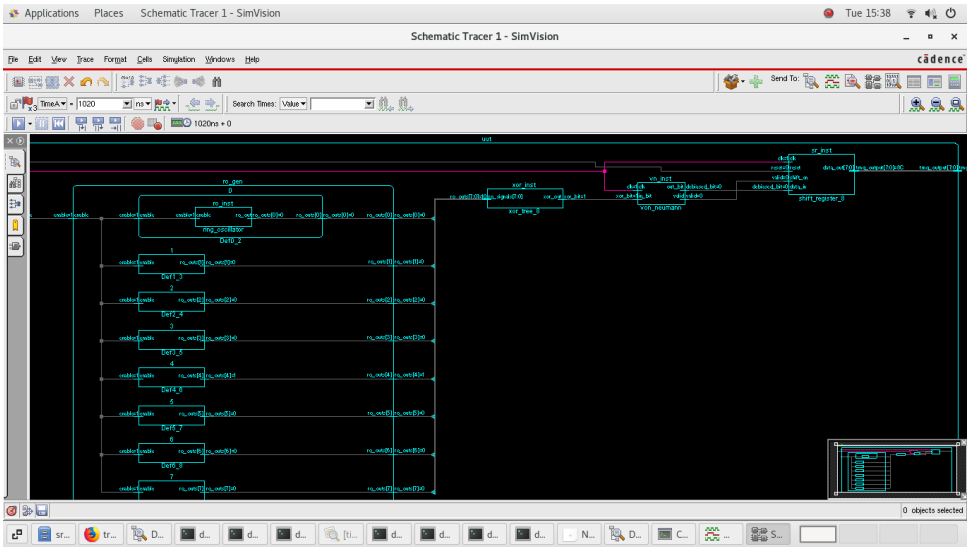


Figure 9: Detailed Circuit Diagram

-

Appendices

Codes

TRNG CODES

```
module trng_top_8bit(
    input clk,
    input reset,
    input enable,
    output [7:0] trng_output
);
    wire [7:0] ro_outs;
    wire xor_bit, debiased_bit, valid;
    genvar i;
    generate
        for (i = 0; i < 8; i = i + 1) begin : ro_gen
            ring_oscillator #(.ID(i)) ro_inst(
                .enable(enable),
                .ro_out(ro_outs[i])
            );
        end
    endgenerate
    xor_tree_8 xor_inst(
        .ro_signals(ro_outs),
        .xor_out(xor_bit)
    );
    von_neumann vn_inst(
        .clk(clk),
        .in_bit(xor_bit),
        .out_bit(debiased_bit),
        .valid(valid)
    );
    shift_register_8 sr_inst(
        .clk(clk),
        .reset(reset),
        .shift_en(valid),
        .data_in(debiased_bit),
        .data_out(trng_output)
    );
endmodule
```

Figure 10: TOP MODULE

```
module trng_tb;

    reg clk = 0;
    reg reset = 0;
    reg enable = 1;
    wire [7:0] trng_output;
    trng_top_8bit uut (
        .clk(clk),
        .reset(reset),
        .enable(enable),
        .trng_output(trng_output)
    );
    always #5 clk = ~clk;

    initial begin
        $dumpfile("trng_output.vcd");
        $dumpvars(0, trng_tb);
        reset = 1;
        #20;
        reset = 0;
        #1000;

        $display("Final TRNG output: %h", trng_output);
        $finish;
    end

endmodule
```

Figure 11: TESTBENCH

```
module ring_oscillator #(parameter ID = 0)(
    input enable,
    output reg ro_out
);

    initial ro_out = 0;

    always begin
        if (enable) begin
            #((ID+1)*3) ro_out = ~ro_out;  // Different toggle delay per ID
        end else begin
            #10; // do nothing when disabled
        end
    end

endmodule
```

Figure 12: SUBMODULE: RING OSCILLATOR

```
module xor_tree_8(
    input [7:0] ro_signals,
    output xor_out
);
    assign xor_out = ^ro_signals; // XOR reduction
endmodule
```

Figure 13: SUBMODULE: XOR TREE

```
module von_neumann(
    input clk,
    input in_bit,
    output reg out_bit,
    output reg valid
);
    reg [1:0] buffer = 2'b00;
    reg toggle = 0;

    always @(posedge clk) begin
        buffer <= {buffer[0], in_bit};
        valid <= 0;
        if (toggle) begin
            if (buffer == 2'b01) begin
                out_bit <= 1'b0;
                valid <= 1;
            end else if (buffer == 2'b10) begin
                out_bit <= 1'b1;
                valid <= 1;
            end
        end
        toggle <= ~toggle;
    end
endmodule
```

Figure 14: SUBMODULE: VON-NEUMANN DEBIASING

```
module shift_register_8(
    input clk,
    input reset,
    input shift_en,
    input data_in,
    output reg [7:0] data_out
);
    always @(posedge clk or posedge reset) begin
        if (reset)
            data_out <= 8'b0;
        else if (shift_en)
            data_out <= {data_out[6:0], data_in};
    end
endmodule
```

Figure 15: SUBMODULE: SHIFT REGISTER

```
read_libs /home/installs/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl trng_fpga.v
elaborate
set_top_module trng_fpga
create_clock -name clk -period 10.0 [get_ports clk]
set_clock_transition -rise 0.1 [get_clocks clk]
set_clock_transition -fall 0.1 [get_clocks clk]
set_clock_uncertainty 0.05 [get_clocks clk]

set_input_delay -max 2.0 -clock clk [get_ports {reset}]
set_output_delay -max 2.0 -clock clk [get_ports {random_number[*]}]
set_false_path -from [get_ports reset]
syn_generic
syn_map
syn_opt
write_hdl > trng_netlist.v
write_sdc > trng_constraints.sdc
report_area   > trng_area.rpt
report_power  > trng_power.rpt
report_timing > trng_timing.rpt
report_gates  > trng_gates.rpt
```

Figure 16: CONSTRAINTS FILE

```
read_libs /home/installs/FOUNDRY/digital/90nm/dig/lib/slow.lib
read_hdl carrylook.v fourbitdffp.v dffp.v topmodule1.v
elaborate
read_sdc constraints.sdc
syn_generic
syn_map
syn_opt
write_hdl > topmodule_netlist.v
write_sdc > output_constraints.sdc
report_area > topmodule_area.rpt
report_power > topmodule_power.rpt
report_timing > topmodule_timing.rpt
report_gates > topmodule_gates.rpt
gui_show
```

Figure 17: SCRIPT.TCL