DESIGN AND IMPLEMENTATION OF PSEUDO RANDOM NUMBER GENERATOR USING VERILOG

A Project Report
Submitted as part of the Academic Internship at the University of Hyderabad
in partial fulfillment of the requirements for the
Bachelor of Technology

BACHELOR OF TECHNOLOGY

In

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

By

P. KARTHIK-2320040019

R. VINAY VARDHAN NAIDU-2320040117

Under the Esteemed Guidance of

Mr. Samrat L. Sabat

हैदराबाद विश्वविद्यालय
University of Hyderabad

Declaration

I hereby declare that the project report entitled "**Design and Implementation of Pseudo random number generator using Verilog**" is a record of original work carried out by me as part of my internship at the University of Hyderabad.

The work presented in this report is authentic and has not been submitted to any other institution or organization for the award of any degree, diploma, or certification. All sources of information and data used in the project have been appropriately acknowledged.

This report reflects the independent contribution made during the internship and complies with the academic integrity standards of project-based learning.

Samrat L. Sabat

P. Karthik – 2320040019

R. Vinay Vardhan Naidu - 2320040117

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to all those who supported and guided me throughout the successful completion of this project titled **"Design and Implementation of Pseudo random number generator using Verilog"**, carried out as part of my internship at the **University of Hyderabad**.

I am deeply thankful to my project mentor, **Samrat L. Sabat** sir, from the Department of Electronics, for his valuable guidance, constant encouragement, and insightful suggestions throughout the duration of the project. His expertise and patience played a crucial role in shaping the outcome of this work.

I would also like to extend my sincere thanks to **Dr. Sunkara Srinivas Rao**, Head of the Department, Electronics and Communication Engineering, for his support and for facilitating this internship opportunity, which significantly enhanced my practical understanding and skills.

I am grateful to the faculty, staff, and peers at the University of Hyderabad and my home institution for their support and assistance. I also acknowledge the contribution of my fellow interns and team members for their collaboration.

.

.

Mr. Samrat L. Sabat

TABLE OF CONTENTS

1. ABSTRACT

Title: Design and Implementation of Pseudo-Random Number Generator Using Verilog

Problem Statement: This project focuses on the design and implementation of a Pseudo-Random Number Generator (PRNG) capable of generating all 65,536 unique (x, y) 8-bit pairs using a deterministic logic-based approach. The objective is to ensure a repeatable, full-period pseudo-random sequence using minimal hardware resources and implement the design on an FPGA platform for verification and testing.

Explanation: Pseudo-random number generators are essential components in cryptography, simulations, test pattern generation, and digital communications. Unlike software-based approaches, this project presents hardware-based PRNG architecture written in Verilog HDL and deployed on an FPGA (Boolean board) for real-time validation. The PRNG core computes new (x, y) pairs from a 16-bit counter, where:

- x[n+1] = (hi + lo) % 256
- y[n+1] = (x[n+1] + lo) % 256

with hi and lo being the upper and lower bytes of the counter respectively.

Design Highlights:

1. **Modular Architecture:** The PRNG is divided into separate Verilog modules:

- counter16: A 16-bit up-counter that generates sequential values from 0 to 65535.
- adder8: An 8-bit adder module used twice to perform (hi + lo) and ((hi + lo) + lo) operations, with wrap-around (modulo-256) behaviour.

2. **Deterministic Logic Generation:** Each (x, y) pair is computed based on:

- x = hi + lo
- y = x + lo
  where hi and lo are the high and low bytes of the 16-bit counter, respectively.

3. **Sequential Output:** The values of x and y are updated on every rising edge of the clock, ensuring that the output progresses deterministically and can be traced step-by-step**.**

4. **Simulation-Friendly Design:** The design includes a step output directly mapped from the internal counter to track the number of generated pairs. This allows easy monitoring in the testbench or waveform viewer.

5. **Compact and Synthesizable:** The design is hardware-friendly, using only basic combinational and sequential elements, making it suitable for implementation on FPGA platforms.

2. INTRODUCTION

In the realm of digital systems, pseudo-random number generators (PRNGs) serve as foundational components in a wide array of applications, including cryptography, error detection, secure data transmission, simulations, gaming, randomized algorithms, and hardware testing. Unlike true random number generators, which rely on unpredictable physical phenomena, pseudo-random number generators produce deterministic sequences that appear random yet are generated using defined logical operations. Their predictability under known initial conditions makes them especially valuable for reproducible testing and simulation environments.

This project presents the design and implementation of a Pseudo-Random Number Generator (PRNG) using Verilog, developed as part of an internship at the University of Hyderabad. The design is synthesized for FPGA implementation, aiming to demonstrate both the practical utility and hardware efficiency of modular pseudo-random number logic. The key objective is to generate unique and repeatable 8-bit (x, y) pseudo-random output pairs over 65,536 steps using simple digital components.

At the core of the system lies the prng_65536 module, which integrates three primary components:

1. **counter16** – A 16-bit synchronous up-counter that continuously increments on each positive edge of the clock, reset asynchronously. It counts from 0 to 65535, providing the seed for generating dynamic pseudo-random values.
2. **adder8** – A reusable 8-bit combinational logic block used to perform modular arithmetic (modulo 256). Two instances of this module are used to transform counter-derived values into pseudo-random output values.
3. **Output Logic** – Registers to hold and output the current values of x and y, updated synchronously with the system clock.

The operation begins with the counter16 module generating a 16-bit count. This value is then split into:

- High byte (hi = counter [15:8])
- Low byte (lo = counter [7:0])

These values are processed through two adders:

- First adder computes x = hi + lo (mod 256)
- Second adder computes y = x + lo (mod 256)

This sequence introduces variability and simulates randomness while remaining deterministic. The use of modular arithmetic ensures all operations stay within 8-bit boundaries, making the outputs suitable for compact, byte-oriented systems. The values of x, y, and the current step count are registered outputs, allowing visualization through simulation tools or waveform viewers like Vivado or ModelSim.

The entire architecture is designed to be lightweight, modular, synthesizable, and suitable for FPGA implementation. It avoids complex feedback shift registers or cryptographic logic,

focusing instead on simplicity and educational value. This makes the design an ideal reference for understanding pseudo-random behaviour at the hardware level and for exploring how predictable structures can produce complex output sequences through arithmetic manipulation.

The project also includes a testbench for validating the design in simulation. Each step of the sequence is monitored and logged, ensuring that all 65,536 unique (x, y) pairs are generated successfully. This not only verifies functional correctness but also provides insight into the deterministic patterns produced by the generator.

In summary, this project demonstrates a clear and efficient hardware implementation of a pseudo-random number generator using modular Verilog design, showcasing the interplay between counter-based sequencing and arithmetic operations to create complex, repeatable output sequences. It lays a strong foundation for further exploration into advanced PRNG algorithms and their applications in secure and embedded systems.

3. LITERATURE SURVEY

As part of my project groundwork, I referred to a research paper that provided valuable insights into the efficient hardware implementation of pseudo-random number generators (PRNGs).

So, the most insightful papers I referred to was titled "**An 8-bit Integer True Periodic Orbit PRNG Based on Delayed Arnold's Cat Map**", authored by Vianney Boniface Ekani Mebengaa, Venkata Reddy Kopparthib, Hermann Djeugoue Nzeugac, J.S. Armand Eyebe Foudac, Guy Morgan Djeufa Dagoumgueic, Georges Bell Bitjokaa, P. Rangababud, Samrat L. Sabatb,. This paper was published in the journal *Computers and Electrical Engineering* by Elsevier in 2023.

The central idea of the paper is to construct a hardware-efficient PRNG based on a modified version of the Arnold's Cat Map (ACM) - a classical chaotic map known for its strong mixing and scrambling properties in image processing and chaos theory. However, traditional chaotic systems like ACM suffer from short-period cycles and precision loss when implemented digitally due to finite word lengths. The authors propose an improved version called the Delayed Quantized Arnold's Cat Map (DQACM) which is fully adapted to 8-bit integer arithmetic and is suitable for FPGA and ASIC designs.

Key Concepts and Formulas:

The paper formulates the delayed ACM in discrete integer space using the recurrence relation:

$$\begin{cases} x_{t+1} = x_{t-\delta} + y_t \\ y_{t+1} = x_{t+1} + y_t \end{cases} \mod 2^n$$

Here:

- x_{t+1}, y_{t+1} are the next state values,
- $x_{t-\delta}$, is a delayed version of the past state by δ\deltaδ steps,
- All computations are done modulo 2^n to preserve 8-bit boundaries (where typically n=8)
- The use of delay δ\deltaδ introduces a longer-period orbit, helping prevent the system from falling into short cycles due to limited precision.

Architecture Overview:

The authors implemented their design on both FPGA (Xilinx Zynq-7000) and 90nm CMOS ASIC platforms. Their design achieved:

- 2.208 Gbps throughput on FPGA
- 5.46 Gbps throughput on ASIC
- Very low area usage and high clock frequency
- Successfully passed all NIST 800-22 statistical tests for randomness

The architecture uses:

- Shift registers to store delayed values
- Arithmetic logic (matrix multiplication)
- Modular operations
- Feedback paths to enhance unpredictability

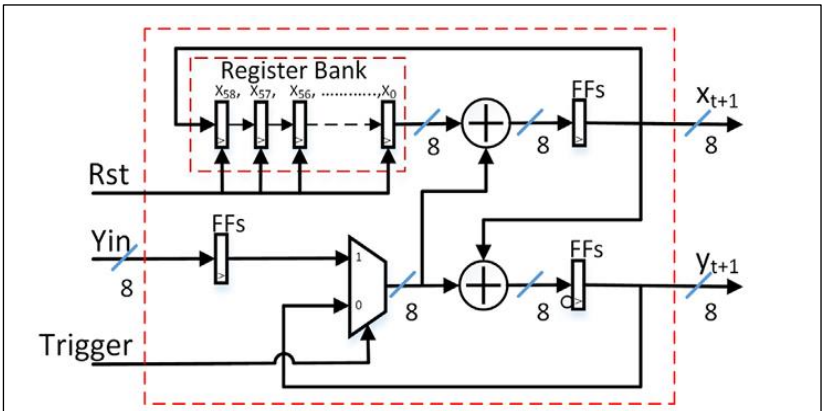


Fig. 1. Hardware architecture of the 8-bit delayed QACM PRNG.

Relevance to My Project:

While my project does not replicate the delayed ACM algorithm directly, studying this paper influenced the core principles behind my Verilog-based PRNG design. I was particularly inspired by:

- The use of 8-bit modular arithmetic to generate pseudo-random values
- The idea of introducing deterministic but non-linear transformations
- The strategy of combining two related values to form a 2D output sequence
- Ensuring outputs remain within 8-bit bounds, making them hardware-friendly

Instead of chaotic maps, my design uses a 16-bit counter whose high and low bytes are fed into two 8-bit adders:

- x = hi + lo (mod 256)
- y = x + lo (mod 256)

This arithmetic-based transformation mimics the spirit of the DQACM: it uses predictable seeds (counter values) but transforms them in such a way that the output appears pseudo-random, all while keeping the design lightweight and suitable for FPGA implementation.

The paper helped me understand that pseudo-randomness can be achieved through simple integer-based transformations without needing complex cryptographic or chaotic systems, especially when the goal is fast, testable, and deterministic output.

4. Software and Hardware Requirements

Hardware Requirements:

1. Laptop/PC Minimum configuration:

- Processor: Intel i5 or higher
- RAM: 8 GB or more
- Storage: 500 GB (SSD recommended)
- Operating System: Windows 10 / Ubuntu 20.04 or above

Software Requirements:

1. **Xilinx Vivado Design Suite**

- Version: 2020.2 or later
- Used for: HDL design entry, synthesis, implementation, bitstream generation, and RTL simulation

2. **ModelSim / Vivado Simulator**

- Used for behavioral simulation and waveform verification of the Verilog modules

3. **MATLAB (optional)**

- Used for validating output patterns and analyzing pseudo-random properties using statistical visualization and histograms

4. **Text Editor / IDE**

- Examples: VS Code, Notepad++, or built-in Vivado editor
- For writing and editing Verilog code

5. Block Diagram

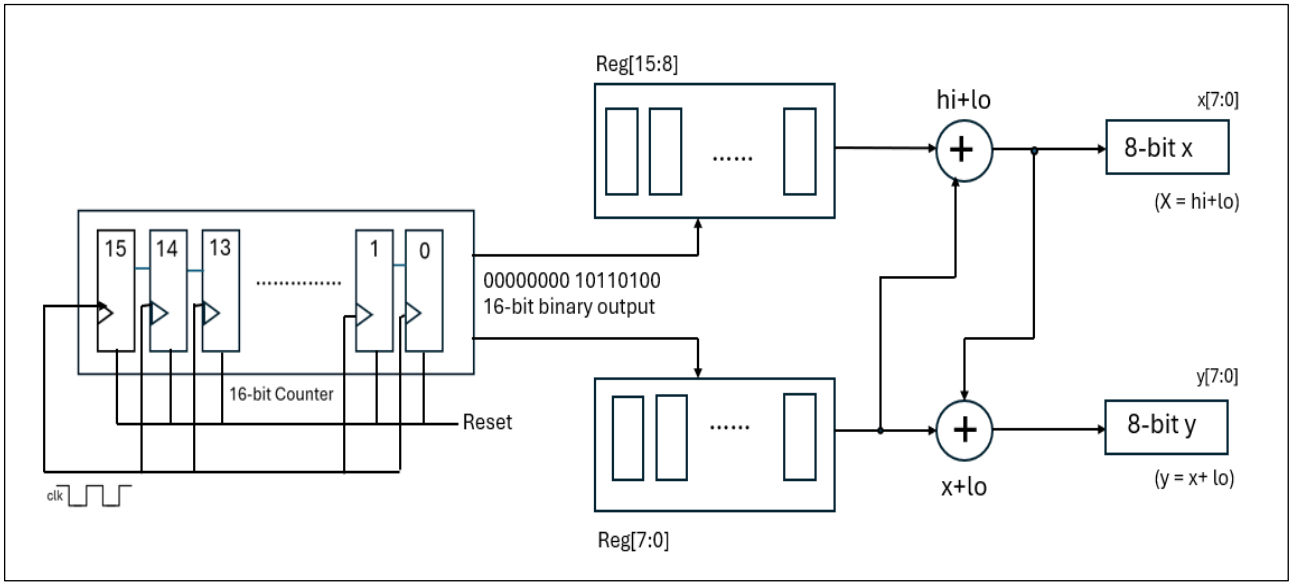


Fig. 2 Block Diagram of PRNG Generator Architecture

The block diagram shown represents the architecture of a Verilog-based Pseudo-Random Number Generator (PRNG) system. It consists of modular components designed for simplicity, clarity, and synthesis on FPGA hardware. The system is driven by a clock and reset signal and produces two 8-bit pseudo-random outputs, x[7:0] and y[7:0], at every clock cycle.

1. 16-bit Counter:

At the core of the design is a 16-bit synchronous up-counter. This counter increments on every rising edge of the clock (clk) and resets to zero when the active-high Reset signal is asserted. The counter output is a 16-bit binary value, which continuously counts from 0 to 65535 and wraps around thereafter.

2. Byte Separation (Reg[15:8] and Reg[7:0]):

The 16-bit counter output is split into:

- **High Byte (Reg[15:8])**: These are the upper 8 bits (bits 15 to 8) of the counter output.
- **Low Byte (Reg[7:0])**: These are the lower 8 bits (bits 7 to 0) of the counter output.

Both of these values are routed to the adder units to participate in arithmetic transformations that generate pseudo-random values.

3. First 8-bit Adder (x = hi + lo):

The first adder performs an 8-bit modular addition of the high and low byte values:

$$x=(hi+lo) \bmod 256$$

This result, x[7:0], is the first pseudo-random output and also serves as an intermediate value for generating y.

4. Second 8-bit Adder (y = x + lo):

The second adder takes the result x from the first adder and adds it again with the low byte value:

$$y=(x+lo) \bmod 256$$

This produces the second pseudo-random output y[7:0].

5. Output:

- **x[7:0]**: First output, derived from high and low byte of counter.
- **y[7:0]**: Second output, derived from x and low byte.

Both outputs update synchronously on each clock cycle, generating a continuous stream of pseudo-random (x, y) pairs.

Therefore,

- On every clock pulse, the counter increments.
- The counter output is split into hi and lo.
- Two adders compute x = hi + lo, and y = x + lo.
- The results are constrained to 8 bits using modulo-256 arithmetic.

This architecture is efficient, modular, and synthesizable on FPGA. It ensures deterministic yet seemingly random output behavior, making it suitable for low-resource PRNG applications in embedded systems and hardware simulations.

6. Working Principle

The pseudo-random number generator (PRNG) implemented in this project follows a simple, deterministic arithmetic logic based on a 16-bit counter and two 8-bit adders. The goal is to generate a sequence of 8-bit (x, y) value pairs that appear random but are completely reproducible and efficient for hardware synthesis.

1. **Clock-Driven Counter Operation:**

- A 16-bit synchronous counter increments on every rising edge of the system clock.
- If the reset signal is active (rst = 1), the counter resets to 0.

2. **Splitting the 16-bit Counter:**

- The current counter value is divided into two 8-bit segments:
- hi = counter[15:8] (upper 8 bits)
- lo = counter[7:0] (lower 8 bits)

3. **First Addition (Generating `x`):**

   The high byte and low byte are added together using modular arithmetic:

   x=(hi+lo) mod256

   This value is stored as the first output, X [7:0]

4. **Second Addition (Generating `y`):**

   The result of the first addition (`x`) is again added to the low byte:

   y=(x+lo) mod256

   This value is stored as the second output, Y [7:0]

5. **Synchronous Output Update:**

- Both x and y are updated on each clock cycle.
- This process continues automatically, generating a new (x, y) pair on every tick of the clock, based on the counter's current value.

6. **Cycle Completion:**

- As the counter is 16 bits, the system produces 65,536 unique combinations before repeating the same sequence.
- This full-period generation makes the design ideal for deterministic random simulation and test applications.

7. Verilog HDL Code

```
`timescale 1ns/1ps

// ================= 16-bit Counter =================
module counter16 (
    input clk,
    input rst,
    output reg [15:0] count
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            count <= 16'd0;
        else
            count <= count + 1;
    end
endmodule

// ================ 8-bit Adder ================
module adder8 (
    input [7:0] a, b,
    output [7:0] sum
);
    assign sum = a + b;  // 8-bit wrap-around (mod 256)
endmodule
```

```
// ================= PRNG Module =================
module prng_65536 (
    input clk,
    input rst,
    output reg [7:0] x,
    output reg [7:0] y,
    output [15:0] step
);
```

```
    wire [15:0] counter;
    wire [7:0] hi = counter[15:8];
    wire [7:0] lo = counter[7:0];

    wire [7:0] sum1, sum2;
    assign step = counter;

    // Instantiate counter
    counter16 cnt_inst (
        .clk(clk),
        .rst(rst),
        .count(counter)
    );

    // Adders
    adder8 add1 (.a(hi), .b(lo), .sum(sum1));
    adder8 add2 (.a(sum1), .b(lo), .sum(sum2));

    // Registers for x and y
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            x <= 8'd0;
            y <= 8'd0;
        end else begin
            x <= sum1;
            y <= sum2;
        end
    end
endmodule
```

```
// ================= Testbench =================
module tb_prng_65536;
    reg clk = 0;
    reg rst = 1;
    wire [7:0] x, y;
    wire [15:0] step;

    prng_65536 uut (
        .clk(clk),
        .rst(rst),
        .x(x),
        .y(y),
        .step(step)
    );

    // Clock generation: 100 MHz
    always #5 clk = ~clk;

    initial begin
        $display("Step\tX\tY");

        #10 rst = 0;

        forever begin
            @(posedge clk);
            $display("%5d\t%3d\t%3d", step, x, y);

            if (step == 16'd70000) begin
                $display("Reached step 70000. Done!");
                $finish;
            end
        end
    end
endmodule
```

The Verilog code for the pseudo-random number generator is written using a mixed modeling style, combining behavioral, structural, and dataflow paradigms. The counter16 and prng_65536 modules use behavioral constructs via always blocks to describe sequential logic such as counters and registers. Structural modeling is employed in the prng_65536 module through the instantiation of submodules (adder8, counter16) to connect functional blocks hierarchically. Additionally, dataflow modeling is used through assign statements for implementing simple combinational logic, such as 8-bit modular addition. This modular and mixed-style approach makes the design both reusable and easy to synthesize on hardware platforms like FPGAs.

8. Architecture

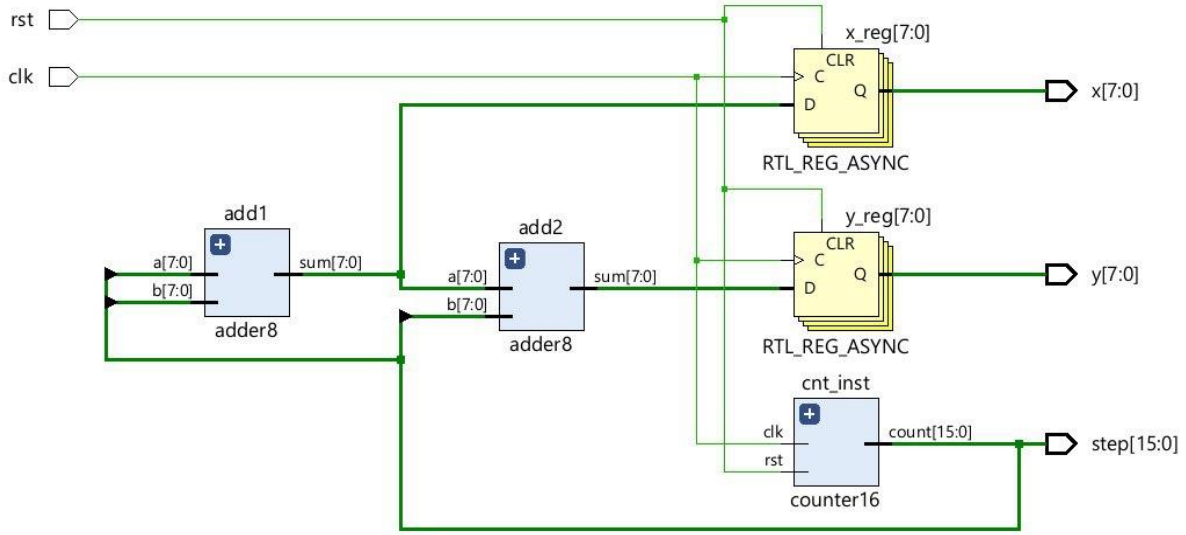


Fig. 3 Architecture of PRNG in Xilinx Vivado

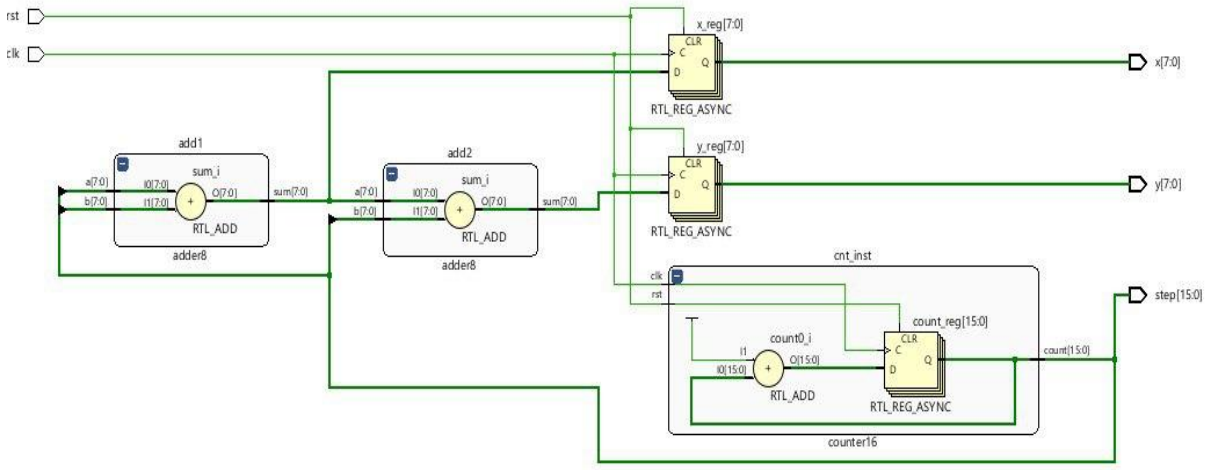
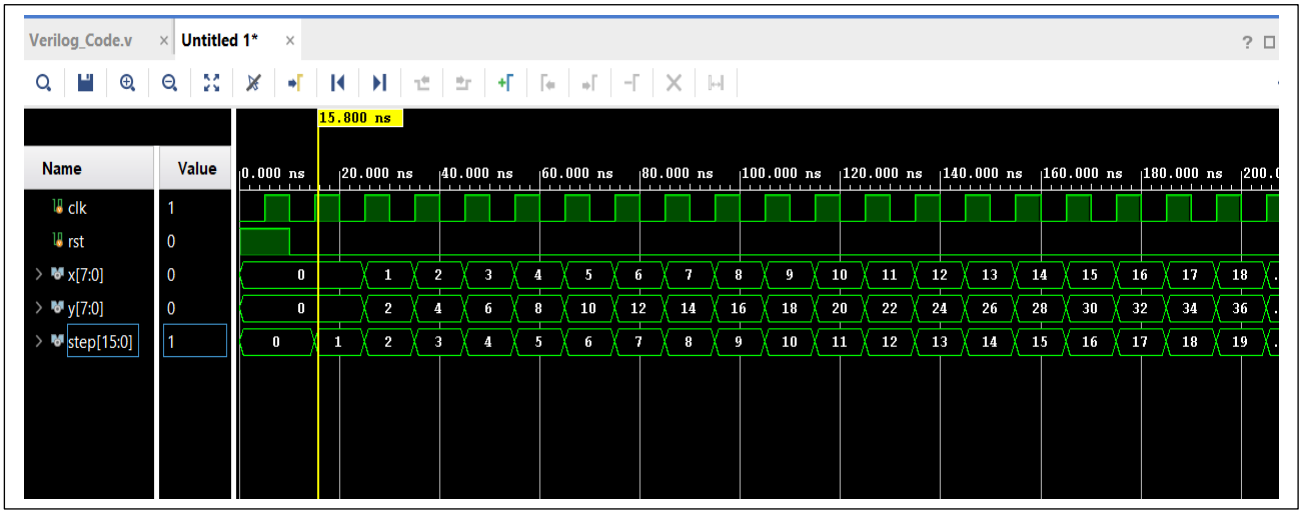
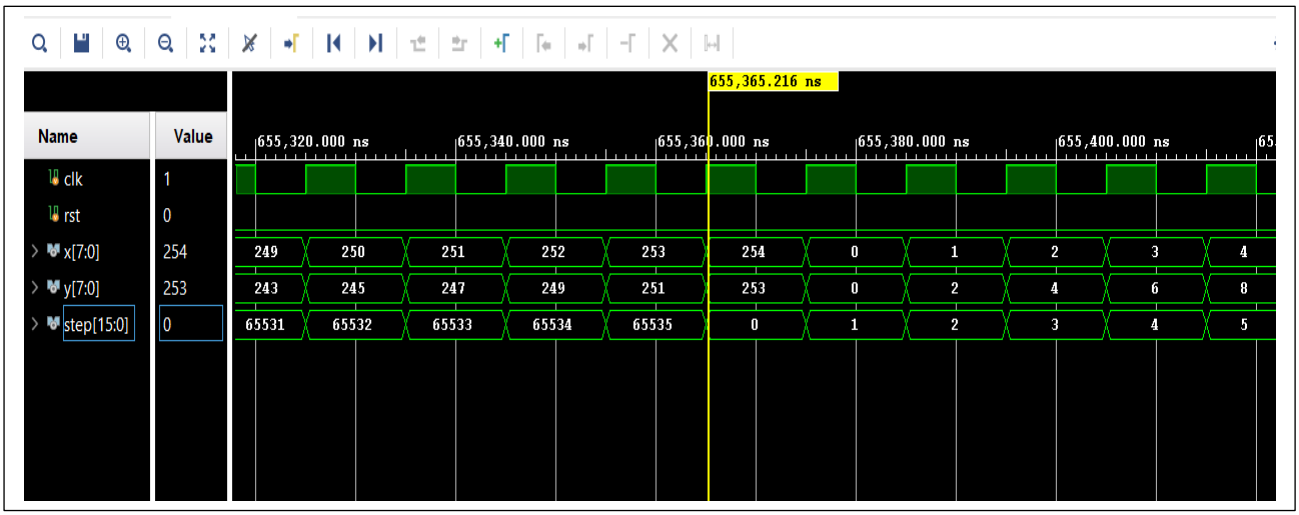


Fig 4. PRNG circuit with internal logic

9. Simulation and Waveforms



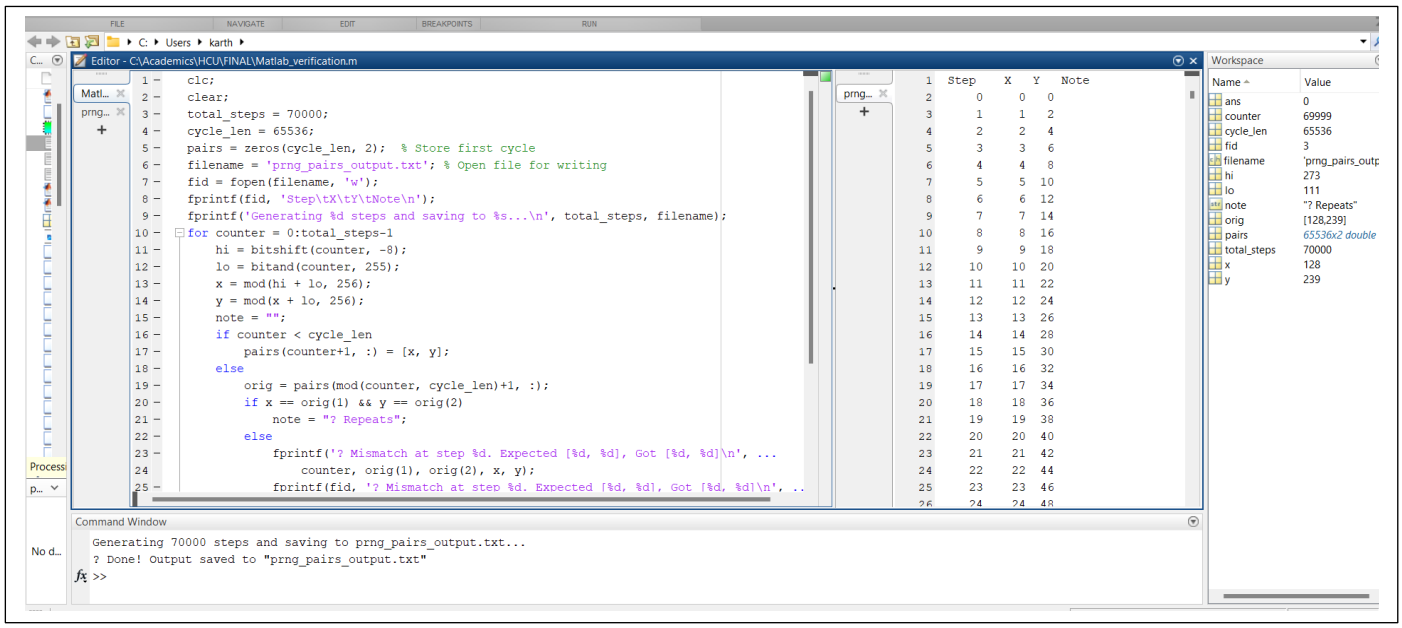


The design was simulated using Vivado's built-in behavioral simulation tools. The testbench (tb_prng_65536) drives the system clock and reset signals, monitors the outputs (x, y, and step), and prints the values for each simulation cycle. The waveform viewer was used to observe and validate the functionality of the pseudo-random number generator (PRNG) module over time.

During simulation, the 16-bit counter continuously increments from 0 to 65535.

After reaching a counter value of 65535, the next value wraps back to 0, causing the sequence of (x, y) values to repeat identically from that point onward. This behavior confirms that the design generates a full cycle of 65,536 unique pseudo-random pairs before repeating, which is the expected result of using a 16-bit counter as the input seed.

10.Output Verification in MATLAB



The script is written in MATLAB language and simulates 70,000 cycles (total_steps = 70000) of PRNG operation.

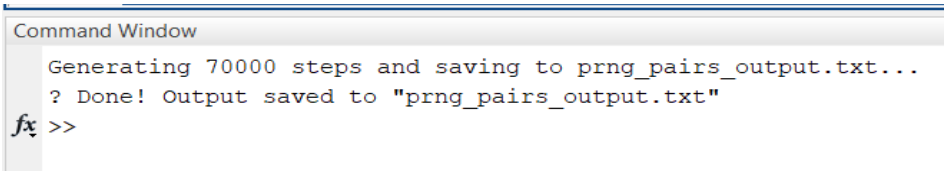
It uses:

- A for loop to iterate through each step
- bitshift and bitand to extract hi and lo byte segments
- Modular arithmetic to compute x = (hi + lo) mod 256 and y = (x + lo) mod 256

- ❖ The first 65,536 values are stored in a pairs array.
- ❖ For steps beyond 65,535, it checks if the (x, y) values repeat the sequence correctly.
- ❖ If values repeat as expected, a note "? Repeats" is tagged.
- ❖ All output is saved to a text file named prng_pairs_output.txt.

MATLAB Command Window (Bottom):

- Displays confirmation messages from the script:



11. Applications / Use Cases

1. Built-In Self-Test (BIST) in VLSI Chips

- PRNGs are used to generate random test patterns for BIST during chip manufacturing.
- This helps detect faults in digital circuits without requiring external test pattern generators.

2. Cryptography and Security Protocols

- While not cryptographically secure by itself, such PRNGs are useful for non-secure operations like nonce generation, session randomization, or as a part of larger cryptographic protocols.
- Hardware PRNGs are used to generate unpredictable sequences for data scrambling and obfuscation.

3. Digital Communication Systems

- In modulation schemes and spread spectrum systems (like CDMA), PRNGs generate pseudo-noise sequences for spreading and de-spreading signals.
- Also used for simulating noise in communication channel models.

4. Simulation and Testbench Automation

- In hardware simulation environments, PRNGs generate test vectors to exercise various logic conditions in a DUT (Device Under Test).
- Useful for regression testing and random stress testing of systems.

5. Gaming and Embedded Entertainment Systems

- Embedded PRNGs are used in game logic for randomness, such as shuffling, random events, and unpredictability in gameplay mechanics.

6. Data Masking and Address Shuffling

- PRNGs help in generating random addresses or data values for use in cache simulations, memory randomization, and address-space layout randomization (ASLR) in security testing.

7. Sensor Noise Simulation

- In signal processing or embedded systems testing, PRNGs can simulate sensor noise or analog interference digitally

12.Conclusion

This project successfully demonstrates the design and simulation of a simple yet effective pseudo-random number generator (PRNG) using Verilog HDL. By combining a 16-bit counter with modular arithmetic through 8-bit adders, the system generates a sequence of 8-bit values (x and y) that simulate randomness across 65,536 steps. Although the design is deterministic, the mathematical structure ensures a varied and non-repeating pattern within the given range, making it suitable for testbench generation, hardware simulations, and learning environments. The project also provided hands-on experience with simulation tools like Vivado, and practical challenges like observing high-speed outputs through FPGA debugging tools such as the ILA. Overall, this work strengthened understanding of counter-based designs, arithmetic logic, and hardware-oriented pseudo-randomness.

13.References

1. V. B. E. Mebenga *et al.*, “An 8-bit integer true periodic orbit PRNG based on delayed Arnold’s cat map,” *AEU - International Journal of Electronics and Communications*, vol. 162, Apr. 2023, doi: 10.1016/j.aeue.2023.154575.

2. GeeksforGeeks Pseudo random number generator

3. Akter, Sonia & Khalil, Kasem & Bayoumi, Magdy. (2024). Efficient Pseudo Random Number Generator (PRNG) Design on FPGA. 1-5. 10.1109/DCAS61159.2024.10539915.