2024 January Term

30.110 Design Systems Lab

Group 3

Name	Student ID
Michelle Ong	1006049
Muhammad Danial Bin Kamalrudin	1006351
Wong Hong Xuan, Shaine	1005540
Zhang Haotian	1006290
Nigel Keng Yi Qian	1005502

1. Design Documentation

- Seed Generation

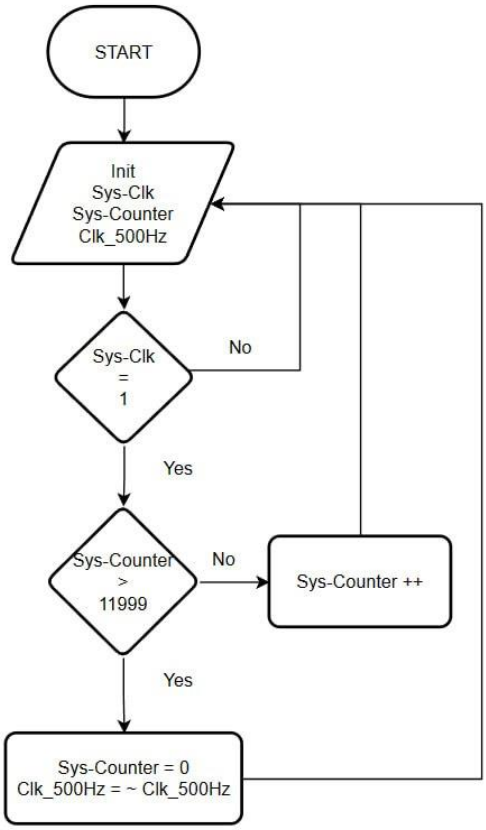
  Our team has utilized an analog sound sensor to capture the variability in sound volumes produced by human activities such as talking and clapping. This approach exploits the natural variability in sound intensity, translating it into seed values for random number generation. This method ensures that at any given time, the input seed value is unique, reflecting the ambient acoustic environment.

- Display

  The design incorporates a display system with a reset functionality that allows users to reinitialize the system as needed. It has a feature time display that is periodically updated to reflect system status and ensures user engagement.

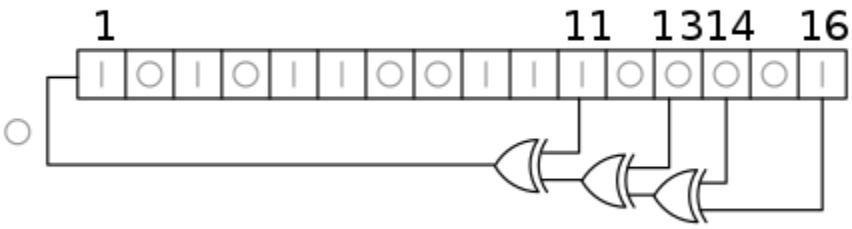
- Clock Tree

  By utilizing a systematic counting approach, we can ensure periodic toggling of the output clock in sync with the system clock. This cycle of counting and resetting maintains the accuracy of the desired frequency but also exhibits the robustness of the clock generation process that is essential for synchronized system operations.



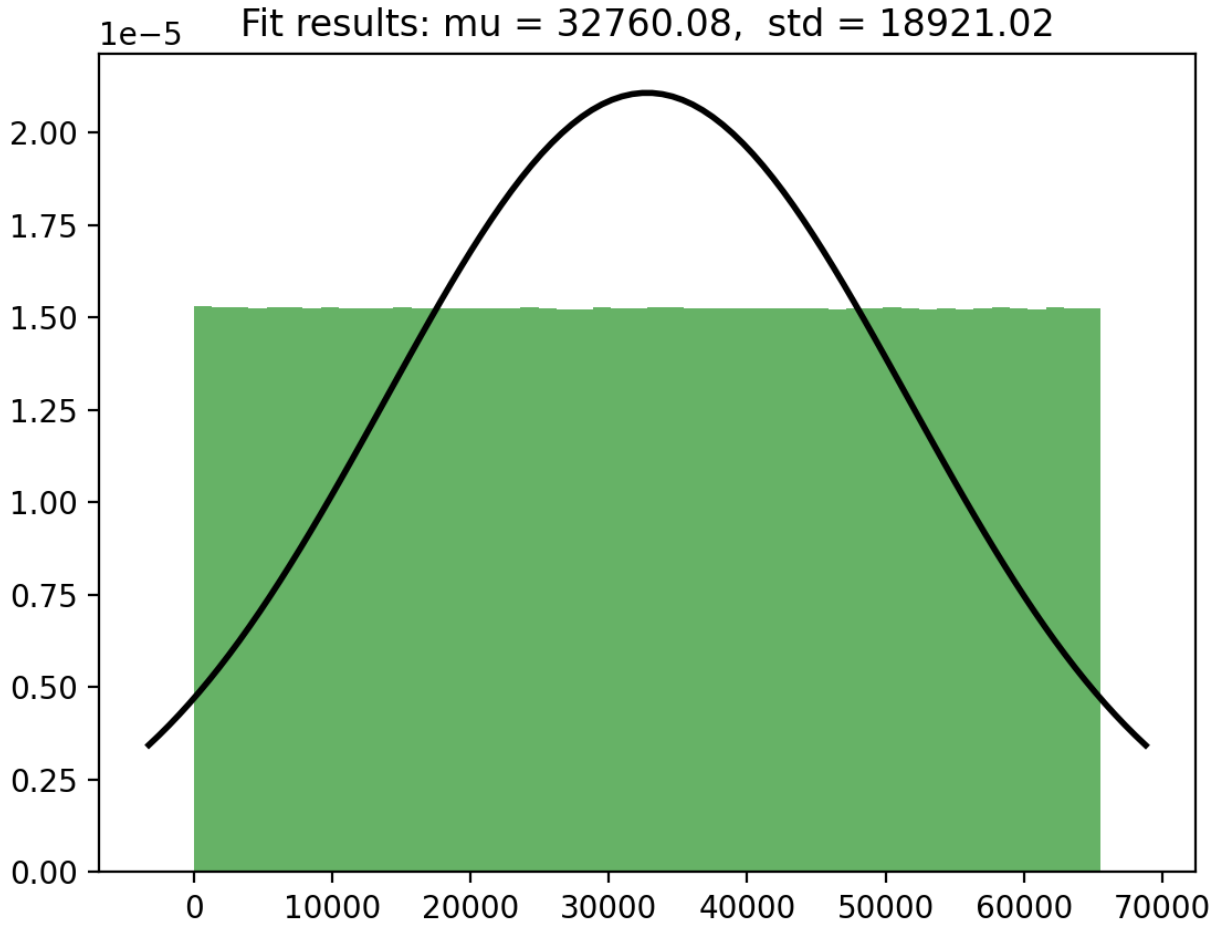
- Pseudo Random Number Generator (PRNG) Design

  Our PRNG is based on a bit shifting algorithm known as the Linear-feedback shift register (LFSR). The LFSR is an important component in our design that influences the randomness and distribution of the output values.



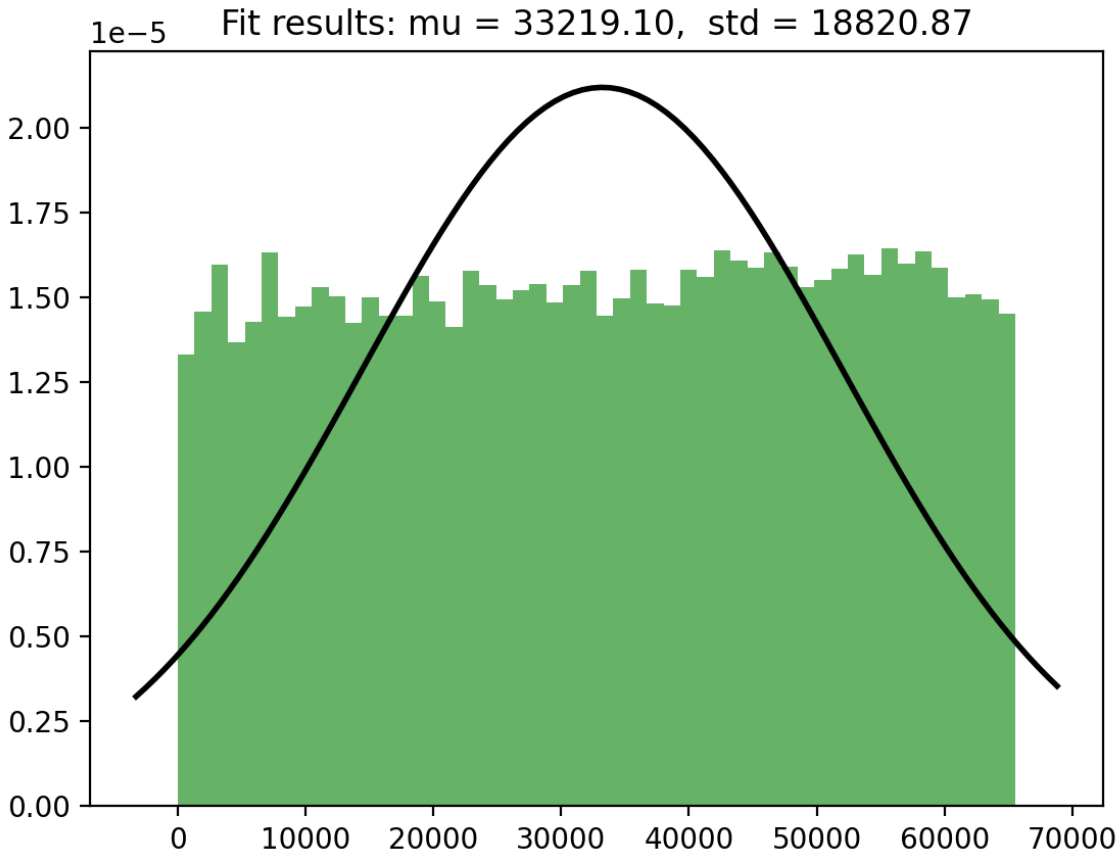
- Testbench

  We then devised a testbench strategy where a single seed will generate 65,535 random numbers. However, the seed will constantly repeat itself resulting in the graph below:



Graph 1: Single seed repeating itself

As such, upon reaching the 65,536th cycle, we increment the original seed by 1 to initiate a new sequence of 65,535 random numbers. This cycle is then repeated in increments of 65,535 to collect an extensive dataset. The graph obtained is shown below:



Graph 2: Seed in increments of 1

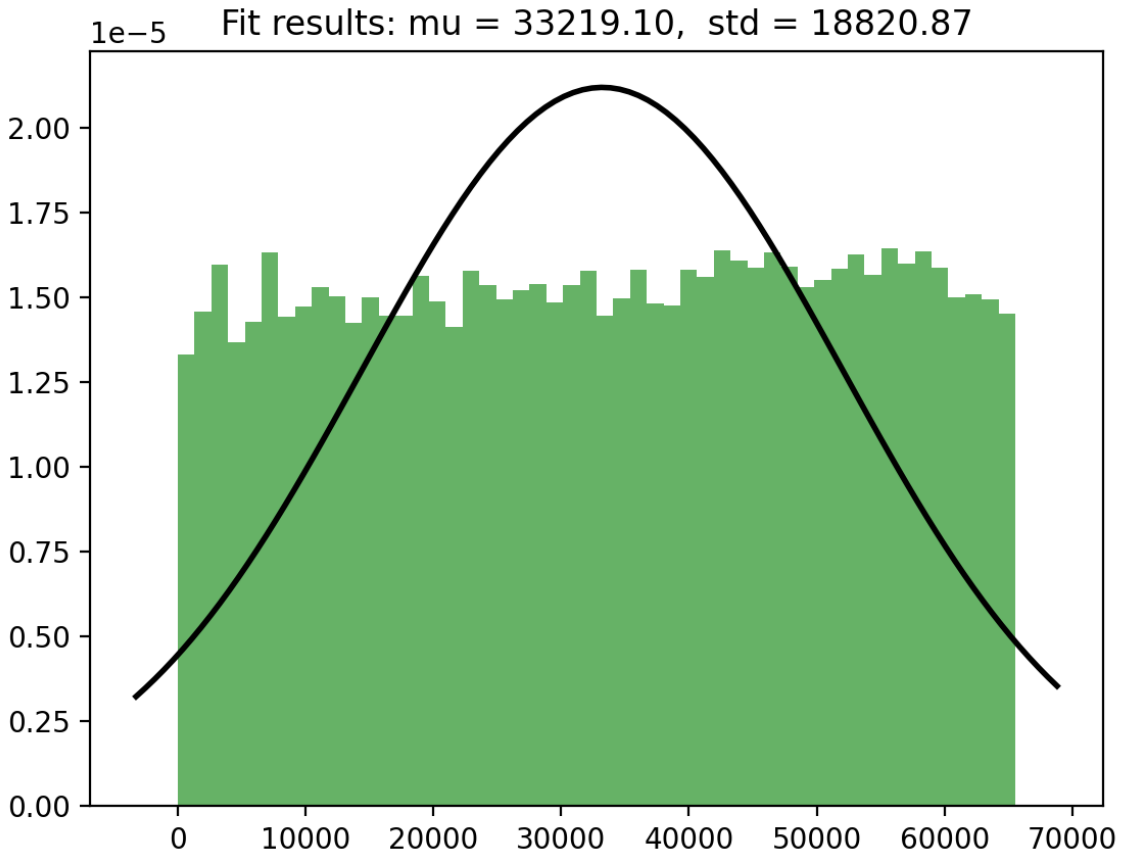
- Data collection from the PRNG

Through the testbench, we accumulated over 1 million data points. These points were collected and stored in a '.txt' file, which serves as the input for our analysis.

- Analysis

A Python script was written to plot the distribution of the generated numbers and evaluate their adherence to the Gaussian distribution. To test for normality, we utilized the `Kolmogorov-Smirnov test for normality`. Below are some main functions used:

- `plt.hist:` this function plots a histogram of the decimal numbers in the dataset
- `mu, std = stats.norm.fit(data['Decimal']):` this function returns mean (mu) and standard deviation (std) of the fitted distribution.
- `stats.norm.pdf(x, mu, std):` this function calculates the PDF of the normal distribution at each point in x, using the previously calculated mu and std
- `unique_count:` this function counts the number of unique numbers in the dataset
-



Graph 2: Seed in increments of 1

The noticeable results indicate that the numbers do not conform to a Gaussian distribution, as seen in the above histogram as it deviates from the normal distribution curve.

2. Challenges encountered

Our initial implementation faced a significant challenge where the PRNG output exhibited repetition after 64 bits, indicating a lack of randomness. To overcome this, we identified and selected a different bit from the shift register, thereby extending the period and enhancing its randomness.

We explored methods to increase randomness and avoid maxed-out number sequences. One approach we did was to modify the initial seed, while another was to introduce a new seed input. Both methods were aimed to disrupt any emergent patterns, thus improving the randomness of the output.

3. Lessons learned

Throughout the implementation process, we have gained insights into the intricacies of random number generation. We learned that even minor alterations in the algorithm or seed value can have profound effects on the randomness and the security of the system.

After consultation, we realized that we did not need XOR but could use an exponential design instead. The figure below is the old code we used:

```
module prng(
    input clk,
    input rst_n,
    input [15:0] seed,
    output reg [15:0] data,
    output reg [15:0] c
);
    reg [15:0] data_next;
    integer i;
    reg reverse_num;

    always @* begin
//         Your combinational logic for data_next
//         Example logic (replace this with your actual logic)
        data_next[15] = data[15] ^ data[1];
        data_next[14] = data[14] ^ data[0];
        data_next[13] = data[13] ^ data_next[15];
        data_next[12] = data[12] ^ data_next[14];
        data_next[11] = data[11] ^ data_next[13];
        data_next[10] = data[10] ^ data_next[12];
        data_next[9]  = data[9]  ^ data_next[11];
        data_next[8]  = data[8]  ^ data_next[10];
        data_next[7]  = data[7]  ^ data_next[9];
        data_next[6]  = data[6]  ^ data_next[8];
        data_next[5]  = data[5]  ^ data_next[7];
        data_next[4]  = data[4]  ^ data_next[6];
        data_next[3]  = data[3]  ^ data_next[5];
        data_next[2]  = data[2]  ^ data_next[4];
        data_next[1]  = data[1]  ^ data_next[3];
```

Figure 1: Old Code

Team Photo: