ENP261 True Random Number Generator

Project Status

Team members: Scotti Bozarth, Alex Wardlow

Summary: Fully implemented true random number generator on FPGA

Contents

Project Goal

The overall goal of this project was to implement a true random number generator. True random number generators have a wide variety of uses; random numbers are widely used in many aspects of computing. Some uses, however, desire the numbers they get to be truly random. A common example of this would be encryption. If you don't want someone to be able to reverse engineer your encryption making it so they can't reverse engineer the random numbers, you use is a good first step. Another possible implementation is if you are making a game to be played with computer scientists who have been known to use the pseudo randomness of the number generator before to gain an edge, as happened in the battleships A.I. contest years ago.

Aspects of Course Content

Finite State Machine

To generate our seed value, we sampled a clock that we used to trigger when we read an internal clock of a different frequency. In order to eliminate bias in our measurements we needed to take two measurements, then process them. This was accomplished through implementing a finite state machine. An illustration of this machine is below.

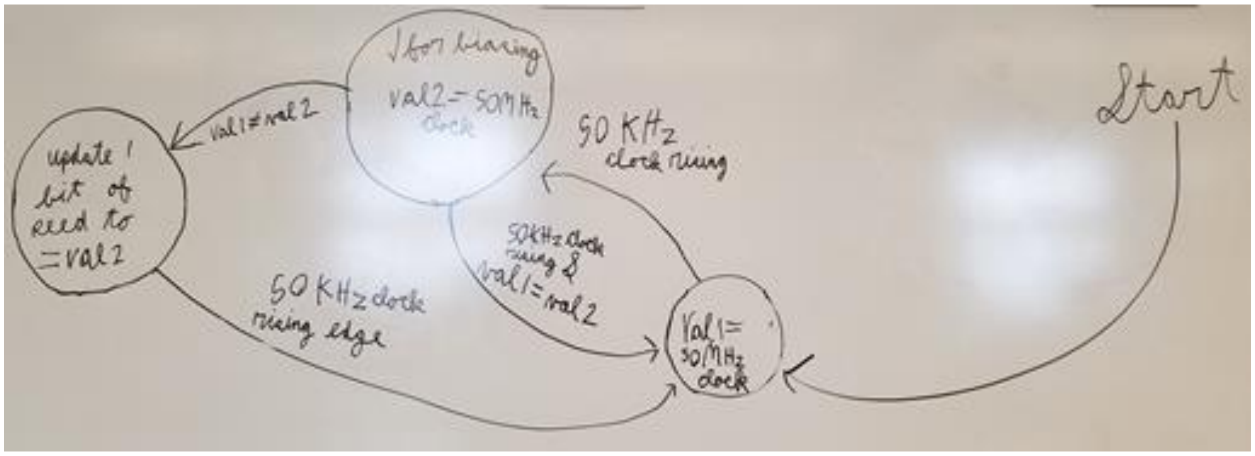

Figure 1 Seedgen Finite State Machine

Synchronous

In order to generate our seed, we looked at the drift of two clocks set to the same frequency. This necessitated measuring the value of one clock, at the rising edge of another. This value was either kept or discarded based on the following table.

M0	M1	Out
0	0	Discard
0	1	0
1	0	1
1	1	Discard

Kept values are temporarily stored until we have 32 of them. When 32 are generated we transfer the lot to a register storing the output, and begin the process anew.

Asynchronous

The seed output is generated pseudo asynchronously. We are unaware of how many samples it will take to generate a new seed, yet as soon as a new seed is generated, it is pushed. The final output does not depend on a set amount of clock cycles, rather the event of generating enough bits to fill a 32 bit value.

FPGA Benefits

The FPGA's unique properties were utilized to minimize the amount of time it takes to turn a seed into a set of random numbers, and the amount of time it takes to process a seed.

Project Guide

Our project was implemented in four major steps, research, seedgen, numbergen, and display.

Research

The first thing we had to tackle was research. There are many different ways to go about generating a true random number generator. For our project we chose to generate a high quality random number using clock drift, then implement the middle square technique to turn this into a high quantity of high quality numbers. Other useful methods to minimize bias and maximize speed were also discovered and implemented.

Seedgen

The seed was generated by measuring the drift of two clocks. The first clock being the FPGA's internal 50 MHz clock. The second coming from a 555 timer IC setup in astable mode as shown below.

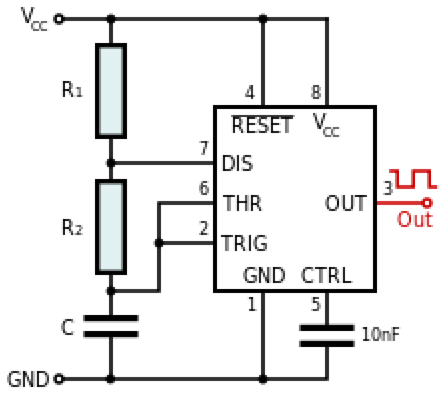

Figure 2 Astable Setup of a 555 Timer IC

Where the frequency of the timer is $\frac{1}{\ln(2)\cdot C\cdot(R_1+R_2)}$. In our case this madeR1 and R2 1MΩ, and C 10nF Unidealistic in our parts made the second clock run at 44±2MHz rather than 50MHz. These unidealisties serve to increase the randomness or our seed.

In order to eliminate any possible bias it is also recommended to take two measurements

M0	M1	Out
0	0	Discard
0	1	0
1	0	1
1	1	Discard

Numbergen

The number gen itself is a simple process to implement. After all the goal of this process it to be able to generate random numbers as quickly as possible, so the less operations the better. For our project we choose to implement the middle square method and add a weyl sequence to it. The best way to illustrate the process is the code, which has been included below.

```
module randomFromSeed(clk, seed, out);
        input clk;
        input [31:0] seed;
        output [15:0] out;
        reg [31:0] x,w;
        initial w = 0;
        initial x = 0;
                always @ (clk) begin
                w <= w+seed;
                x <= w+(x*x);
        end
        assign out = x[23:8];
endmodule
```

This code generates a 16 bit random number very quickly compared to our seedgen. The addition of the weyl sequence along with ensuring parity of the lsb of the seed prevents our random number from both collapsing to zero, and entering a repeating sequence. In other words, a lot of research and thought was put into making the process appear this simple.

Display

The original plan was to transmit the numbers to the computer, and display them that way. In the end we decided that this was outside the scope of the project and implemented a binary to hex display that would show one of the random numbers generated every 1/6th of a second.

Final Result

The final design is not showing any signs of not working. In other words, it appears to be working as intended. It is impossible to prove that any random number generator is truly random. But our number generator follows all the practices the community has deemed necessary to earn that moniker. In other words, it is suitably random. Better than many default implementations of random number generators.

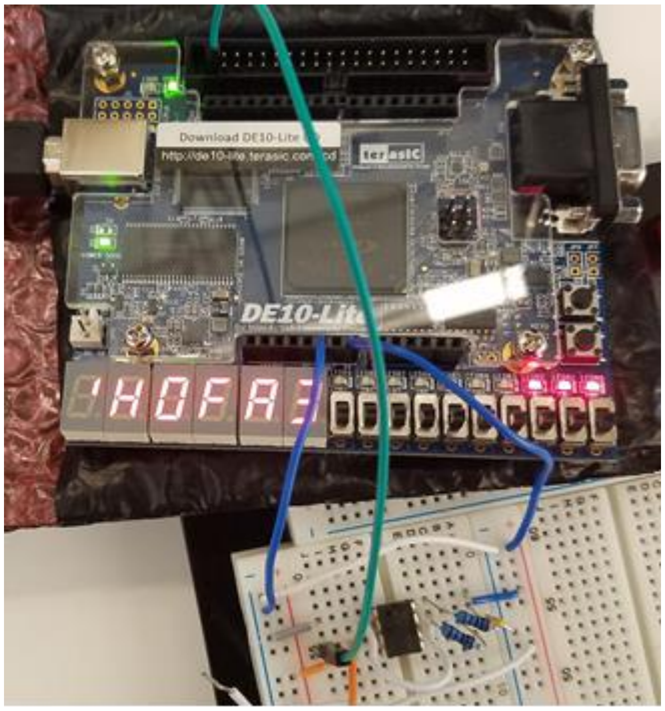

Figure 3 Full Hardware Implementation of True Random Number Generator