JISSE
ISSN: 2636-4425

Journal of International Society for Science and Engineering
Vol. 5, No. 1, 08-14 (2023)

JISSE
E-ISSN:2682-3438

FPGA Design and Implementation for Pseudorandom Number Generator based square root (SR-PRNG)

Ghada Elsayed [1,*], Somaya Kayed[2]

[1] *Assistant Professor, Electrical Department, MTI University, Egypt*

[2] *Associate Professor, Electrical Engineering Department, Obour High Institute for Engineering and Technology, Egypt*

A R T I C L E I N F O

Article history:
Received:16-01-2023
Accepted:10-03-2023
Online:01-03-2023

Keywords:
FPGA
MATLAB HDL CODER
VHDL

A B S T R A C T

Randomness has been used as a seed for cryptographic algorithms and wireless communication protocols, and today, it is used as a tool or a feature in data preparation that maps input data to output data to make predictions in machine learning. Randomness importance is our research motivation. We introduce an efficient FPGA Design and Implementation for a mathematical calculation of the square root of irrational numbers. We adopted this Square Root Pseudo-Random Number Generator (SR-PRNG). This SR-PRNG is already designed, simulated, and tested for randomness. The advantage of FPGA implementation is to use it as a module in a modular design either in cryptographic applications or machine learning applications. We propose First Order Recursive Generation and Second-order Recurrence Generation design and FPGA implementation. We compare their utilizations for the target FPGA. The target FPGA is Xilinx Spartan 6 XC6SLX4-2CPG196. MATLAB HDL Coder is used for the design. The Maximum frequency is 244.499MHz for the two designs. The utilization of the second-order design versus the first-order design is 262 vs. 169 in the Number of Slice Registers, 368 vs. 207 in the Number of Slice LUTs, 227 vs. 133 in the Number of fully used LUT-FF pairs, 2 vs. 1 in the Number of Block RAM/FIFO, however, they are equal in the Number of bonded IOBs and the Number of BUFGs.

1. Introduction

Today and tomorrow are the eras of artificial intelligence and machine learning. One of the most important parts of success is testing. When the machine is huge and has many Predictors (Inputs) and consequently has huge samples (observation) it would be very difficult to cover the whole scenario. The use of randomness is an obligation here. In addition, PRNG (Pseudo Random Number Generator) has been used for cryptographic algorithms and wireless communication protocols. We already have worked in both eras in which we need PRNG implemented over an FPGA. We adopted the Square Root Pseudo-Random Number Generator (SR-PRNG) algorithm. This algorithm uses the decimal places of irrational numbers. First Order Recursive Generation and Second-order Recurrence Generation are expressed by equation (1) and equation (2).

$$x_{i+1} = \sqrt{x_i} \bmod 1 \times 10^n \qquad (1)$$

$$x_{i+1} = \sqrt{x_i + x_{i-1}} \bmod 1 \times 10^n \qquad (2)$$

It takes the square root of the current n-digit number and the first n digits in its decimal place are the next number, where, x_0 is not a perfect square number, and n is the number of digits needed to generate each iteration. Tests show a disadvantage of the first order equation (1). There is an absorbing state with a frequency of $1/\sqrt{10^n}$. The second order was presented to avoid absorbing states and periods. The overhead of the second order in the software implementation was speed [1]. This method was published and tested for the two basic empirical tests of statistical randomness: the equidistribution test and the serial test [1] by the department of mathematics Rose-Hulman institute of technology. In section 2 we discuss the related works. Next, in section 3 we

* Ghada Elsayed, Electrical Department, MTI University, Egypt, +201002337149, ghada.farouk@eng.mti.edu.eg

present the workflow of designing FPGAs using MATLAB HDL Coder. . In section 4 we propose the hardware design and implementation to it. In section 5 we discuss the testing result. Finally, we highlighted the conclusion.

2. Related Works

Many PRNGs are implemented over FPGA for example “Reconfigurable SR-PRNG pseudo-random number generator based on FPGA” [2]. They also adopted and implemented an already published and tested algorithm. Their target was XC5VLX50T FPGA which is larger than our target. Their implementation has utilized 1.4% and 16.7% of the available slices and DSP blocks respectively. The Maximum frequency was 78 MHz which is much slower than ours. They didn't use MATLAB HDL CODER. There are many ways for the PRNG, the most famous one is by chaotic [3], [4], and [5], Elliptic curves can be found in [6] and [7]. An example of implementing PRNG using for design is “'FPGA Design and Implementation for Adaptive Digital Chaotic Key Generator “[3]. They also adopted and implemented an already published and tested algorithm. They use the same targeted FPGA. The Maximum frequency was 15.711MHz. The area utilization was 189 slice registers, 2303 slice LUTs (lookup tables), 126 fully used LUT, 68 IOs, and 16 DSP blocks. Our utilization is smaller than their utilization for the same target except, we use approximately double the number of slice registers. There are many other implementations of different PRNGs over FPGA for many applications like [8], and [9].

3. Fpga Design Using Matlab Hdl Coder

HDL Coder™ software enables the high-level design of FPGAs, SoCs, and ASICs by generating portable, composable Verilog® and VHDL® code from MATLAB® functions, Simulink® models, and Stateflow® diagrams. The generated HDL code can be used for FPGA programming, ASIC models, and production design [10]. Its workflow is shown in Figure 1. This paper is based on a Matlab script. Matlab HDL Coder project uses two files; the first one is the top-level function of the design and the second one is its test. Then the HDL CODER converts this script to an HDL file. The HDL file is then forwarded to the integrated FPGA synthesizer. The generation of the HDL code from the MATLAB script requires understanding the I/O types. This is necessary for the FPGA pins. It also requires making all types compatible with each other. All the used functions and libraries should be mapped into hardware. Functions like printf and scanf are examples of not supported functions. The Matlab HDL Coder as a tool for FPGA Design was a subject for evaluation by many researchers. They first recommended using it for only fast proofing of the idea. This was the result of area utilization and speed compared to the VHDL is not encouraging [11]. They compare the two methods, the VHDL method versus the MATLAB HDL CODER method. The number of Slice Registers in the VHDL path is 49% versus 8% in MATLAB coder which is much higher because of using the pipeline method. But the Number of Slice LUTs is 40% in VHDL while it exceeds the maximum in MATLAB which is 410 percent. According to timing, the Clock frequency in VHDL was a Maximum Frequency: 120.279MHz while in MATLAB HDL Coder Maximum Frequency: 24.353MHz. However, the latest research has proposed an Optimization technique to dramatically, improve that result [12]. They compared the MATLAB HDL Coder by itself with and without the proposed optimization technique. The Number of Slice LUTs requirements improved from 366% to 72%. The frequency improved from 26.574MHz to 185.355MHz. They proposed a technique for optimizing loops and introducing pipelining at the same time. Based on that recommendation we here design and implement the generation over the FPGA using MATLAB HDL Coder.

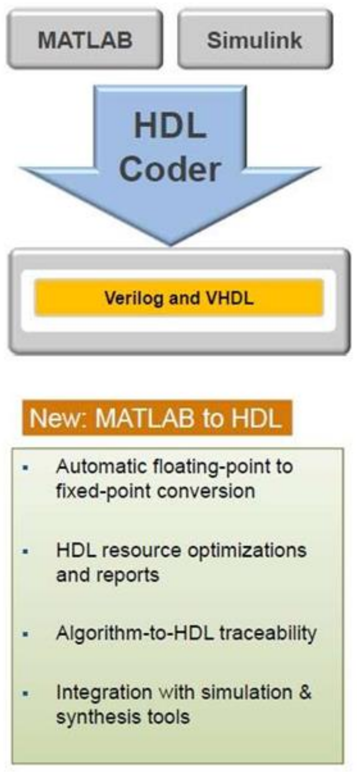
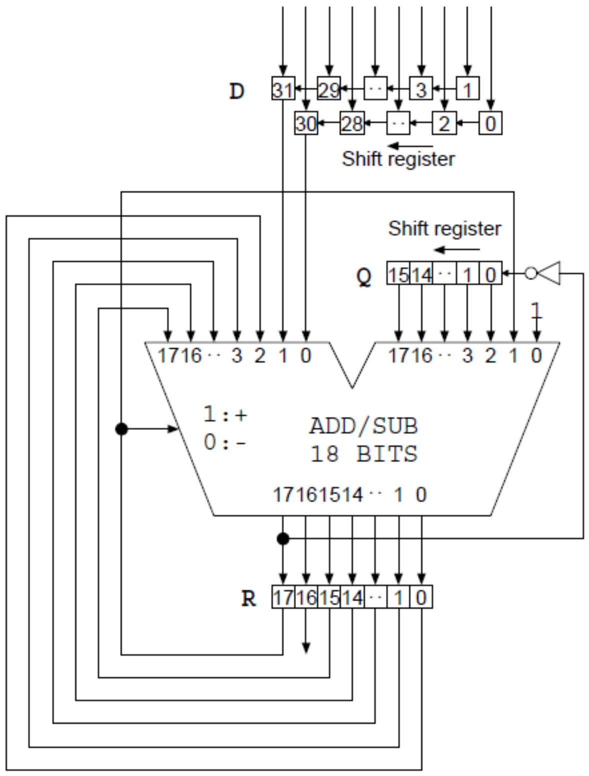

MATLAB
Simulink
HDL Coder
Verilog and VHDL
New: MATLAB to HDL
Automatic floating-point to fixed-point conversion
HDL resource optimizations and reports
Algorithm-to-HDL traceability
Integration with simulation & synthesis tools

Figure 1: MATLAB HDL Coder Workflow [10]

In the following section, we will propose the FPGA Design from bottom to top scripts for the first and second-order designs.

4. Fpga design and implementation of square root calculation

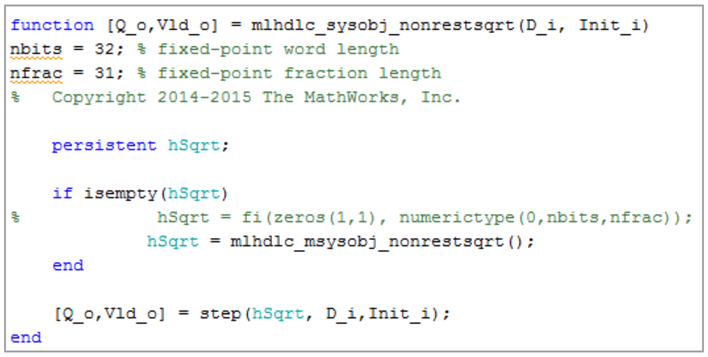
MathWorks introduces a very good script for calculating the square root as an example for MATLAB HDL Coder [13]. HDL Code Generation from A Non-Restoring Square Root System Object is an example that shows how to check, generate and verify HDL code from MATLAB® code that instantiates a non-restoring square root system object. We start with this example as a core of our design. Figure 2 shows the architecture of this example. Figure 3 illustrates the Matlab script for this example and its tester is shown in Figure 4.



[13]

Figure 2: Non-Restoring Square Root Script overall architecture

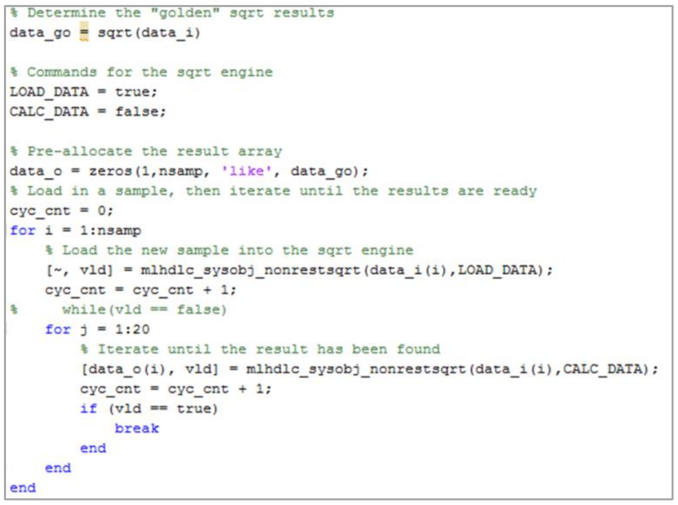
The feedback in figure 2 is implemented in the code using the persistent keyword in figure 3. There are two outputs one is logical. When it becomes true it means the output is ready after certain iteration. The other is the square root output. Also, the input has one logical. Even if it is false means that it's loading time otherwise, it iterates and takes the feedback instead of the initial loaded value



```
function [Q_o,Vld_o] = mlhdlc_sysobj_nonrestsqrt(D_i, Init_i)
nbits = 32; % fixed-point word length
nfrac = 31; % fixed-point fraction length
%   Copyright 2014-2015 The MathWorks, Inc.

    persistent hSqrt;

    if isempty(hSqrt)
%             hSqrt = fi(zeros(1,1), numerictype(0,nbits,nfrac));
             hSqrt = mlhdlc_msysobj_nonrestsqrt();
    end

    [Q_o,Vld_o] = step(hSqrt, D_i,Init_i);
end
```

Figure 3: Non-Restoring Square Root Script

This function needs to be within for loop that makes the iteration. This is illustrated in the tester code of Figure 4. The tester loop for nsamp generates nsamp PRNG output words. Inside each cycle of the tester loop there exist 20 times loop for iteration to get one output PRNG word. The number 20 is achieved by an empirical method.



```
% Determine the "golden" sqrt results
data_go = sqrt(data_i)

% Commands for the sqrt engine
LOAD_DATA = true;
CALC_DATA = false;

% Pre-allocate the result array
data_o = zeros(1,nsamp, 'like', data_go);
% Load in a sample, then iterate until the results are ready
cyc_cnt = 0;
for i = 1:nsamp
    % Load the new sample into the sqrt engine
    [~, vld] = mlhdlc_sysobj_nonrestsqrt(data_i(i),LOAD_DATA);
    cyc_cnt = cyc_cnt + 1;
%     while(vld == false)
    for j = 1:20
        % Iterate until the result has been found
        [data_o(i), vld] = mlhdlc_sysobj_nonrestsqrt(data_i(i),CALC_DATA);
        cyc_cnt = cyc_cnt + 1;
        if (vld == true)
            break
        end
    end
end
```

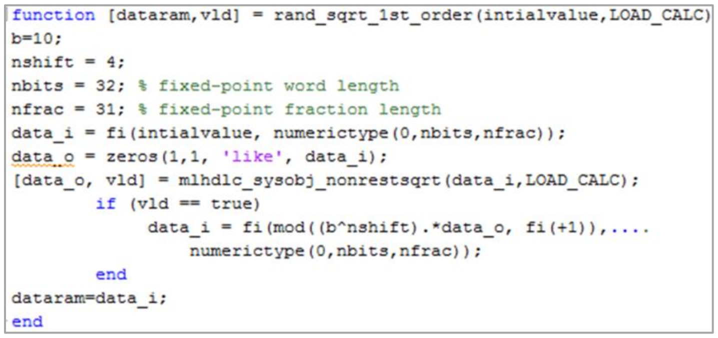
Figure 4: Test for non-restoring root script

In the following section, we will propose the design by MATLAB script to the First Order Recursive Generation code then its tester results, and its rtl top-level view. This will be repeated in the Second-order Recurrence Generation code.

5. Random number generator using matlab hdl coder for fpga design res

In this section we present the First Order Recursive Generation and Second-order Recurrence Generation consequently, we here illustrate the code and its tester without and with storage. We should mention here that our design is reconfigurable because we can just change the generic values for the base number "b", Number of shifts "nshifts", fixed point word length "nbits", and fixed point fraction length "nfrac".

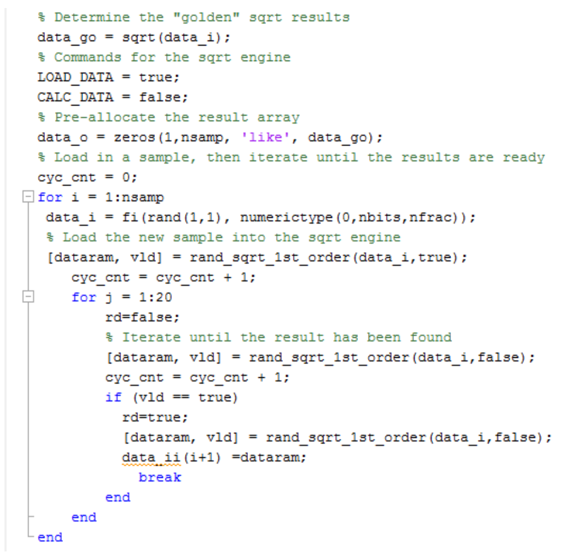
Figure 5 illustrates the code for the first order SR-PRNG equ(1). We used the fixed-point numeric object to unify almost all the types. Inside the condition that tests that the output is ready; i.e. vld is true, the calculation of equ (1) is performed.



```
function [dataram,vld] = rand_sqrt_1st_order(intialvalue,LOAD_CALC)
b=10;
nshift = 4;
nbits = 32; % fixed-point word length
nfrac = 31; % fixed-point fraction length
data_i = fi(intialvalue, numerictype(0,nbits,nfrac));
data_o = zeros(1,1, 'like', data_i);
[data_o, vld] = mlhdlc_sysobj_nonrestsqrt(data_i,LOAD_CALC);
        if (vld == true)
              data_i = fi(mod((b^nshift).*data_o, fi(+1)),....
                    numerictype(0,nbits,nfrac));
        end
dataram=data_i;
end
```

Figure 5: SR-PRNG Script for first-order design

Here we call the MathWorks example [13] and then apply equation (1) to the example's output. The tester of this design is shown in Figure 6, taking into consideration the type casting.

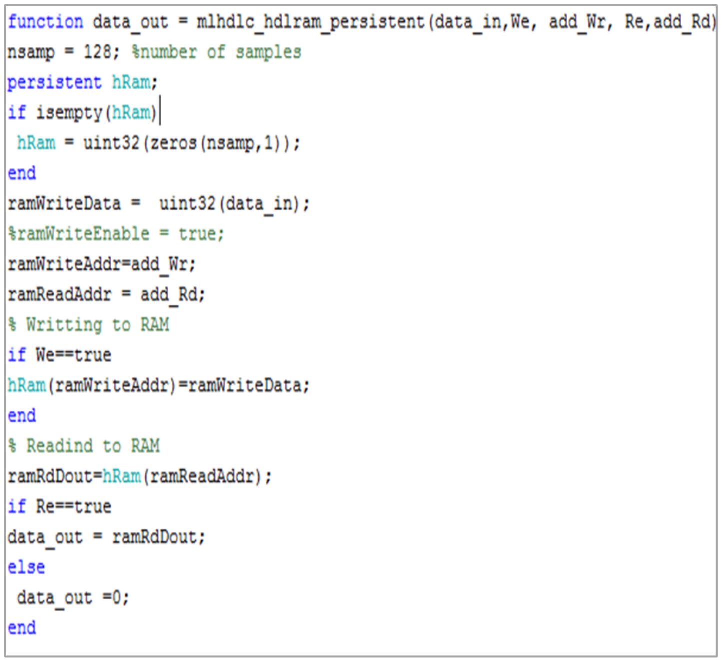


```
% Determine the "golden" sqrt results
data_go = sqrt(data_i);
% Commands for the sqrt engine
LOAD_DATA = true;
CALC_DATA = false;
% Pre-allocate the result array
data_o = zeros(1,nsamp, 'like', data_go);
% Load in a sample, then iterate until the results are ready
cyc_cnt = 0;
for i = 1:nsamp
 data_i = fi(rand(1,1), numerictype(0,nbits,nfrac));
 % Load the new sample into the sqrt engine
 [dataram, vld] = rand_sqrt_1st_order(data_i,true);
    cyc_cnt = cyc_cnt + 1;
    for j = 1:20
        rd=false;
        % Iterate until the result has been found
        [dataram, vld] = rand_sqrt_1st_order(data_i,false);
        cyc_cnt = cyc_cnt + 1;
        if (vld == true)
          rd=true;
          [dataram, vld] = rand_sqrt_1st_order(data_i,false);
          data_ii(i+1) =dataram;
            break
        end
    end
end
```

Figure 6: Test for SR-PRNG Script for first-order design

In the tester of Figure 6, we first call the rand_sqrt_1st for loading by setting the load/Calc signal to true, then call it many times with load/Calc be false until the vld output turn true. At this time we take the output of the function.

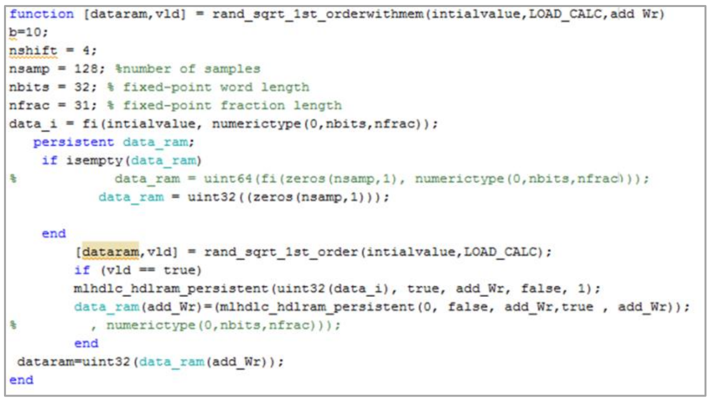
Then we want to store the output random number in memory and plot the stored value. The script of the memory is shown in Figure 7, the design with memory is shown in Figure 8, the tester for that design is shown in Figure 9, the output plot is shown in figure 10, and the top view of the design is shown in figure 11.



```
function data_out = mlhdlc_hdlram_persistent(data_in,We, add_Wr, Re,add_Rd)
nsamp = 128; %number of samples
persistent hRam;
if isempty(hRam)
 hRam = uint32(zeros(nsamp,1));
end
ramWriteData =  uint32(data_in);
%ramWriteEnable = true;
ramWriteAddr=add_Wr;
ramReadAddr = add_Rd;
% Writting to RAM
if We==true
hRam(ramWriteAddr)=ramWriteData;
end
% Readind to RAM
ramRdDout=hRam(ramReadAddr);
if Re==true
data_out = ramRdDout;
else
 data_out =0;
end
```

Figure 7: Matlab script for memory design

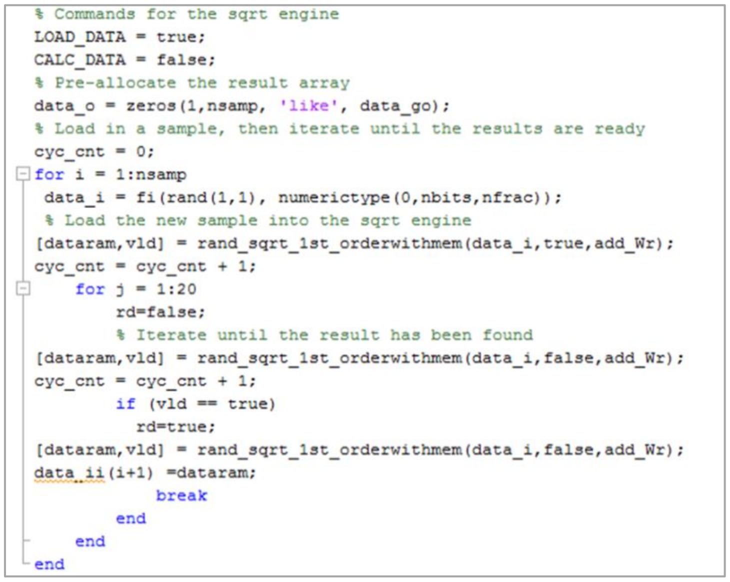
To implement memory we can write the code of Figure 7. You can find memory examples in [13].



```
function [dataram,vld] = rand_sqrt_1st_orderwithmem(intialvalue,LOAD_CALC,add Wr)
b=10;
nshift = 4;
nsamp = 128; %number of samples
nbits = 32; % fixed-point word length
nfrac = 31; % fixed-point fraction length
data_i = fi(intialvalue, numerictype(0,nbits,nfrac));
   persistent data_ram;
    if isempty(data_ram)
%            data_ram = uint64(fi(zeros(nsamp,1), numerictype(0,nbits,nfrac)));
          data_ram = uint32((zeros(nsamp,1)));

    end
        [dataram,vld] = rand_sqrt_1st_order(intialvalue,LOAD_CALC);
        if (vld == true)
        mlhdlc_hdlram_persistent(uint32(data_i), true, add_Wr, false, 1);
        data_ram(add_Wr)=(mlhdlc_hdlram_persistent(0, false, add_Wr,true , add_Wr));
%         , numerictype(0,nbits,nfrac)));
        end
 dataram=uint32(data_ram(add_Wr));
end
```

Figure 8: SR-PRNG Script for first order design with output stored in memory

The diffraction here is that we just add two lines for writing and reading the memory.



```
 % Commands for the sqrt engine
 LOAD_DATA = true;
 CALC_DATA = false;
 % Pre-allocate the result array
 data_o = zeros(1,nsamp, 'like', data_go);
 % Load in a sample, then iterate until the results are ready
 cyc_cnt = 0;
for i = 1:nsamp
  data_i = fi(rand(1,1), numerictype(0,nbits,nfrac));
  % Load the new sample into the sqrt engine
 [dataram,vld] = rand_sqrt_1st_orderwithmem(data_i,true,add_Wr);
 cyc_cnt = cyc_cnt + 1;
     for j = 1:20
         rd=false;
         % Iterate until the result has been found
 [dataram,vld] = rand_sqrt_1st_orderwithmem(data_i,false,add_Wr);
 cyc_cnt = cyc_cnt + 1;
         if (vld == true)
           rd=true;
 [dataram,vld] = rand_sqrt_1st_orderwithmem(data_i,false,add_Wr);
 data_ii(i+1) =dataram;
             break
         end
     end
 end
```

Figure 9: Tester for first order SR-PRNG with memory

The output plot in figure 10 shows that the output varies randomly. However, the randomness tests are already performed within a thesis at the Department of Mathematics Rose-Hulman Institute of Technology [1].

Figure 11 illustrates that for generating 32-bit PRNG and storing up to 256 words of it. We need 80 I/O pins. 8 bits for a memory address, 32 bits for the input non-square number, and 32 big for the output PRNG. We need also 4 input bits for the clock, the clock reset, and the Load/Calc control signal. We need also two output bits for indicating that the output is calculated successfully. The area utilization of the target FPGA and the maximum frequency are discussed in section 5.

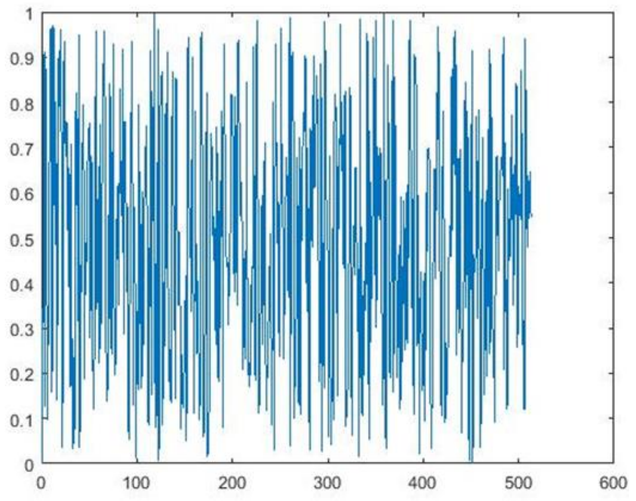


Figure 10: Plot of the output of the first-order SR-PRNG output

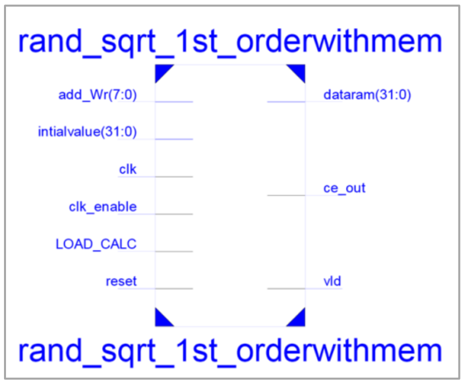
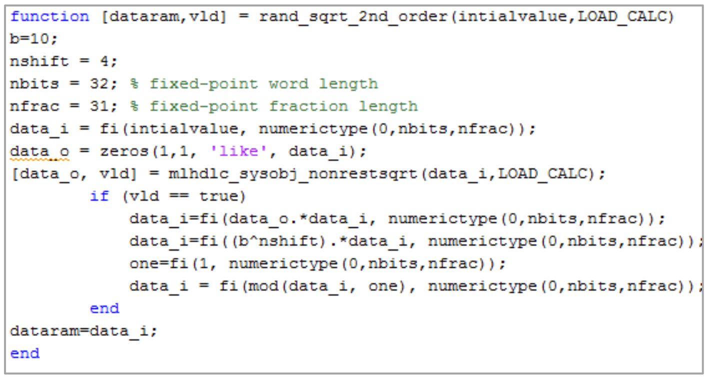


Figure 11: Top view of first-order SR-PRNG design

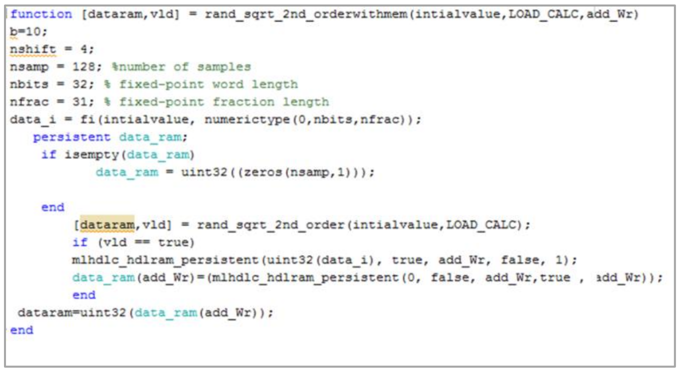
The second-order design SR-PRNG of equation (2) script is shown in Figure 12, the design with memory is shown in Figure 13, the second order SR-PRNG with memory tester is shown in Figure 14, the plot of the output of the second-order design is shown in Figure 15 and the top level view of the second order design is shown in Figure 16.



```
function [dataram,vld] = rand_sqrt_2nd_order(intialvalue,LOAD_CALC)
b=10;
nshift = 4;
nbits = 32; % fixed-point word length
nfrac = 31; % fixed-point fraction length
data_i = fi(intialvalue, numerictype(0,nbits,nfrac));
data_o = zeros(1,1, 'like', data_i);
[data_o, vld] = mlhdlc_sysobj_nonrestsqrt(data_i,LOAD_CALC);
        if (vld == true)
            data_i=fi(data_o.*data_i, numerictype(0,nbits,nfrac));
            data_i=fi((b^nshift).*data_i, numerictype(0,nbits,nfrac));
            one=fi(1, numerictype(0,nbits,nfrac));
            data_i = fi(mod(data_i, one), numerictype(0,nbits,nfrac));
        end
dataram=data_i;
end
```

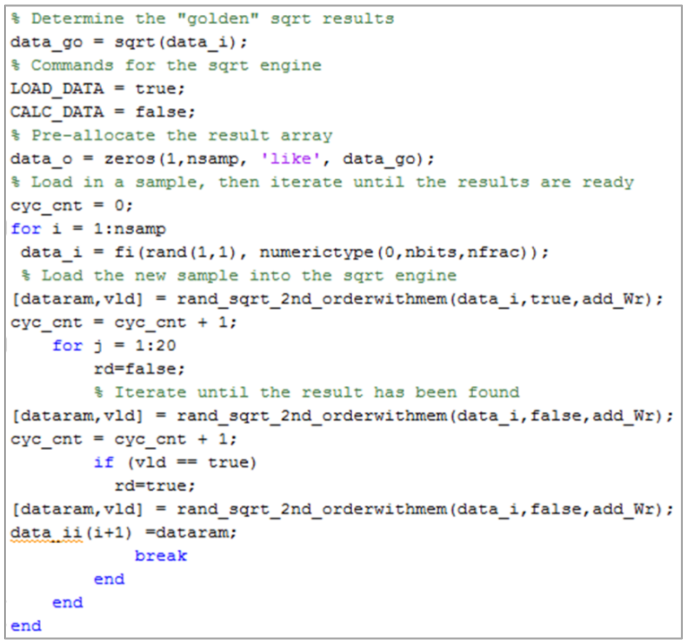
Figure 12: SR-PRNG Script for second order design

In this function, we just replace the lines calculate equation (1) by equation (2). This is to overcome the weakness of absorption and periods [1].



```
function [dataram,vld] = rand_sqrt_2nd_orderwithmem(intialvalue,LOAD_CALC,add_Wr)
b=10;
nshift = 4;
nsamp = 128; %number of samples
nbits = 32; % fixed-point word length
nfrac = 31; % fixed-point fraction length
data_i = fi(intialvalue, numerictype(0,nbits,nfrac));
   persistent data_ram;
    if isempty(data_ram)
          data_ram = uint32((zeros(nsamp,1)));

    end
        [dataram,vld] = rand_sqrt_2nd_order(intialvalue,LOAD_CALC);
        if (vld == true)
        mlhdlc_hdlram_persistent(uint32(data_i), true, add_Wr, false, 1);
        data_ram(add_Wr)=(mlhdlc_hdlram_persistent(0, false, add_Wr,true , add_Wr));
        end
 dataram=uint32(data_ram(add_Wr));
end
```

Figure 13: SR-PRNG Script for second order design with output stored in memory



```
% Determine the "golden" sqrt results
data_go = sqrt(data_i);
% Commands for the sqrt engine
LOAD_DATA = true;
CALC_DATA = false;
% Pre-allocate the result array
data_o = zeros(1,nsamp, 'like', data_go);
% Load in a sample, then iterate until the results are ready
cyc_cnt = 0;
for i = 1:nsamp
 data_i = fi(rand(1,1), numerictype(0,nbits,nfrac));
 % Load the new sample into the sqrt engine
[dataram,vld] = rand_sqrt_2nd_orderwithmem(data_i,true,add_Wr);
cyc_cnt = cyc_cnt + 1;
    for j = 1:20
        rd=false;
        % Iterate until the result has been found
[dataram,vld] = rand_sqrt_2nd_orderwithmem(data_i,false,add_Wr);
cyc_cnt = cyc_cnt + 1;
        if (vld == true)
          rd=true;
[dataram,vld] = rand_sqrt_2nd_orderwithmem(data_i,false,add_Wr);
data_ii(i+1) =dataram;
            break
        end
    end
end
```

Figure 14: Tester for second order SR-PRNG with memory

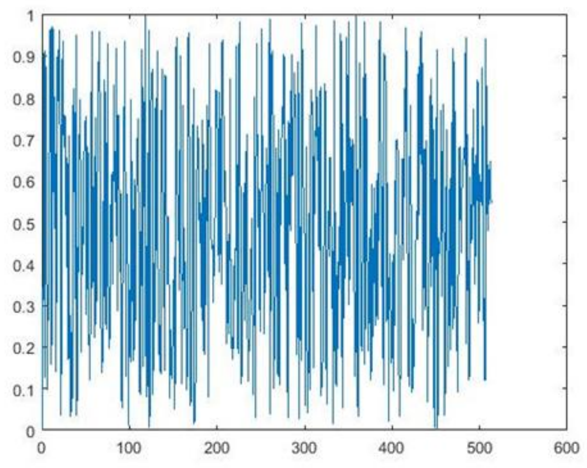


Figure 15: Plot of the output of the second-order SR-PRNG output

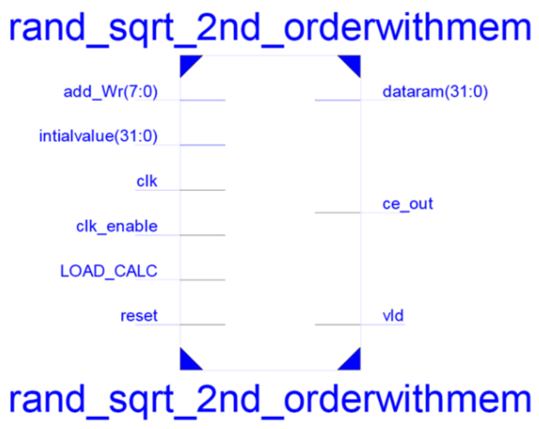


Figure 16 Top view of second-order SR-PRNG design.

Figure 16 illustrates that for generating 32-bit PRNG and storing up to 256 words of it we need 80 I/O pins. 8 bits for a memory address, 32 bits for the input non-square number, and 32 big for the output PRNG. We need also 4 input bits for the clock, clock enable reset, and Load/Calc control signal. We need also two output bits for indicating that the output is calculated successfully. The area utilization of the target FPGA and the maximum frequency are discussed in section 5

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

The authors would like to acknowledge Electronics Research Institute (ERI), Egypt for supporting us with the simulation tool (MATLAB).

6. RESULTS

The Target Device is Xilinx Spartan 3 xc6slx9-2-cpg196 FPGA [14]. The estimated values for device utilization for the first-order design and the second-order design are illustrated in table 1 and a histogram illustration for the two designs is illustrated in figure 17. The Maximum frequency for the first and the second order designs are the same which is 244.499MHz.

Table 1: Device Utilization Summary (estimated values)

Logic Utilization	Utilization for 1st order	Utilization for 2nd order
Number of Slice Registers	1%	2%
Number of Slice LUTs	3%	6%
Number of fully used LUT-FF pairs	54%	56%
Number of Block RAM/FIFO	3%	6%
Number of BUFG/BUFGCTRL/BUFHCEs	6%	6%

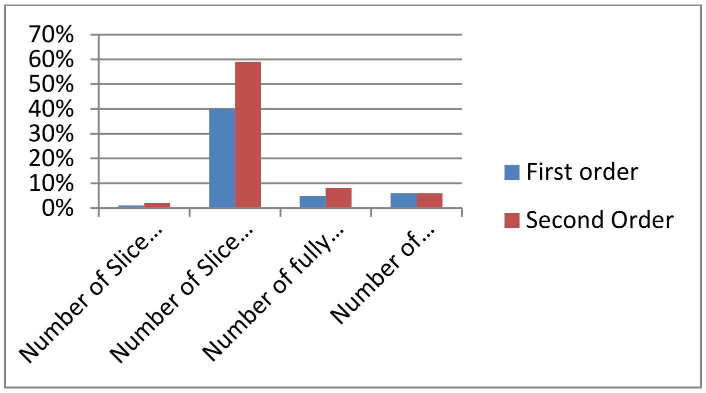


Figure 17 Top view of second-order SR-PRNG design.

The utilization of the first order design is the Number of Slice Registers 262, the Number of Slice LUTs is 368, No. of fully used LUT-FF pairs is 227 Number of bonded IOBs is 80. No. of Block RAM/FIFO is 2, No. of BUFG/BUFGCTRL/BUFHCEs 1. The utilization of the second order design Number of Slice Registers is 169, the Number of Slice LUTs is 207 Number of fully used LUT-FF pairs is 133, the Number of bonded IOBs 48, the Number of Block RAM/FIFO is 1, and the Number of BUFG/BUFGCTRL/BUFHCEs 1

7. Conclusion

In this paper, we proposed a design and implementation of the first and second-order SR-PRNG generators over the FPGA. The target device was xc6slx9-2-cpg196. The design and its implementation were done by using MATLAB HDL Coder. The target FPGA was Xilinx Spartan 3 xc6slx9-2-cpg196. We introduced the two designs. The Maximum frequency is 244.499MHz for the two orders which are very fast compared to the other in the literature. The utilization of the second order versus the first order designs is 262 vs 169 in the Number of Slice Registers,368 vs 207 in the Number of Slice LUTs, 227 vs 133 in the Number of fully used LUT-FF pairs, 80 vs 80 in the Number of bonded IOBs, 2 vs 1 of Number of Block RAM/FIFO, and 1 vs 1 of Number of BUFG/BUFGCTRL/BUFHCEs. We proposed a reconfigurable and very efficient PRNG design in terms of both speed and Area utilization.

References

[1] Su, J. and McSweeney, J. (2019) Square Root Pseudo-Random Number Generators. thesis. Department of Mathematics Rose-Hulman Institute of Technology.

[2] Ahmed A. Rezk a, Ahmed H. Madian d, Ahmed G. Radwan c,a, Ahmed M. Soliman b, "Reconfigurable SR-PRNG pseudo random number generator based on FPGA", Int. J. Electron. Commun. (AEÜ) 98 (2019) 174–180.

[3] Elsayed, G., Kayed, S. (2022). 'FPGA Design and Implementation for Adaptive Digital Chaotic Key Generator', AEAS 49th annual conference, https://aeas2022.asu.edu.eg.

[4] Sambas, A. et al. (2022) “A novel 3D chaotic system with line equilibrium: Multistability, Integral Sliding Mode control, electronic circuit, FPGA implementation and its image encryption,” IEEE Access, 10, pp. 68057–68074. Available at: https://doi.org/10.1109/access.2022.3181424..

[5] Tutueva, A.V. et al. (2020) “Adaptive SR-PRNG maps and their application to pseudo-random numbers generation,” Chaos, Solitons &amp; Fractals, 133, p. 109615. Available at: https://doi.org/10.1016/j.chaos.2020.109615..

[6] AbdElHaleem, S.H., Abd-El-Hafiz, S.K. and Radwan, A.G. (2022) “A generalized framework for elliptic curves based PRNG and its utilization in image encryption,” Scientific Reports, 12(1). Available at: https://doi.org/10.1038/s41598-022-17045-x.

[7] Syafalni, I. et al. (2022) “Efficient homomorphic encryption accelerator with integrated PRNG using low-cost FPGA,” IEEE Access, 10, pp. 7753–7771. Available at: https://doi.org/10.1109/access.2022.3143804.

[8] Al-Musawi, W.A., Wali, W.A. and Al-Ibadi, M.A. (2021) “Implementation of SR-PRNG system using FPGA,” 2021 6th Asia-Pacific Conference on Intelligent Robot Systems (ACIRS) [Preprint]. Available at: https://doi.org/10.1109/acirs52449.2021.9519360.

[9] DRIDI, F. et al. (2021) “Design, FPGA-based implementation and performance of a pseudo random number generator of chaotic sequences,” Advances in Electrical and Computer Engineering, 21(2), pp. 41–48. Available at: https://doi.org/10.4316/aece.2021.02005.

[10] Eda techchannel OpenSystems Media. Available at: http://tech.opensystemsmedia.com/eda/2012/03/mathworks-introduces-hdl-coder-and-verifier-for-matlab/ (Accessed: January 8, 2023).

[11] Elsayed, G., Kayed, S. (2022). 'A Comparative Study between MATLAB HDL Coder and VHDL for FPGAs Design and implementation', Journal of International Society for Science and Engineering, 4(4), pp. 92-98. DOI: 10.21608/jisse.2022.136645.1056 available at https://jisse.journals.ekb.eg/article_260808.html

[12] Kayed, S., Elsayed, G. (2022). 'Optimizing Techniques for using MATLAB HDL CODER', AEAS 49th annual conference, https://aeas2022.asu.edu.eg

[13] MathWorks, HDL code generation from a non-restoring square root system object, HDL Code Generation from A Non-Restoring Square Root System Object - MATLAB &amp; Simulink. Available at: https://www.mathworks.com/help//hdlcoder/ug/hdl-code-generation-from-a-non-restoring-square-root-system-object.html (Accessed: January 15, 2023).

[14] Cmod S6™ FPGA Board Reference Manual, Revised June 13, 2017. 1300 Henley CourtPullman, WA 99163 509.334.6306 www.digilentinc.com

Abbreviation and symbols

BUFG	**Global buffer**
FPGA	**Field programmable Gate Arrays**
HDL	**Hardware Description Language**
LUT	**Look Up Tables**