Parallel Random Number Generator

Noel J Loxton[†] Keenan Robinson[‡] Mauro G Borrageiro[§]
EEE4120F Class of 2020
University of Cape Town
South Africa
[†]LXTNOE001 [‡]RBNKEE001 [§]BRRMAU002

Abstract—**The purpose of this paper is to investigate a pseudo-random number generation technique known as Linear Feedback Shift Registers (LFSR). This random number generator is implemented using a standard serial program coded in C^{++} and subsequently produced in a hardware-accelerated version of the algorithm, which would be run on a Field Programmable Gate Array (FPGA), namely the NEXYS A7 Artix 100-t from Digilent®. The latter is implemented using a form of parallel programming in Verilog Code and simulation. The two implementations are tested and compared to evaluate process speed-up. Further details regarding statistical tests are given, which are performed to ensure optimal performance of the algorithm. The digital accelerated parallel implementation of the LFSR random number generator produced random data-sets with huge speed-ups compared to the Golden Standard. The data-sets were acceptably random but their entropy needs to be improved in order to widen the applications of this parallel random number generator.**

I. Introduction

The modern technological world relies on a number of tools, algorithms and scientific knowledge to act as ground work for new innovations and discoveries to advance into the future. An appropriate tool used in a variety of different applications of digital systems is the need for random numbers, more specifically the random number generator (RNG).

The universe is a fairly unpredictable place and humans have always had a desire to understand it better through scientific understanding and the use of powerful tools such as simulations and modern modelling technology.

To model the randomness on these systems, there needs to be some technology, which can create 'randomness'. The idea of 'randomness' is fairly foreign to computer based systems, which are entirely logic based, albeit, over time different techniques have been developed to simulate the human idea of randomness [1]. Producing sets of truly random numbers is fairly simple however it is surprising that they do not hold as much value as being able to produce randomness in a controlled manner. To be more specific, it is better to be able to produce randomness that can be understood and reproduced yet seeming 'random' to the user for the application it is being applied to. This is called "pseudo-randomness."

For the purposes of this research, the Pseudo-Random Number Generator (PRNG) that is being implemented is the Linear Feedback Shift Register (LFSR). The reason for this was due to its ability to maximise random number production before repeating the sequence [2], which is a desirable aspect among random number generators, hardware realisability, which makes it easily reproducible on architecture systems such as the Field Programmable Gate Array (FPGA) - the hardware that will be used to produce a digitally accelerated version of the LFSR PRNG.

With reference to the real world applications of these pseudo-random numbers (PRNs), generally applications can require extremely large amounts of PRNs to be produced at a time, where serial implementations can start to produce adverse effects on processing speeds. Thus, as a plausible solution to this need, architectures that accommodate parallel computation are used to produce the data required. The FPGA can therefore be used to provide the hardware resources to implement this design variation, although, it is limited due to its size available circuit elements. For the purposes of this research, a NEXYS A7 Artix 100-t FPGA board from Digilent®was used to test the Parallel Random Number Generator (PPRNG) produced in Verilog to compare to the C^{++} serial version. This serial program will be the Golden Standard.

The one aspect of the design is to initially produce PRNs on the different architectures and, from observation, observe the randomness of the outputs. The next step after this design is to further improve the design to provide suitable random numbers statistically, so that it provides good entropy in the values being produced with a uniform distribution. This method is often used to assess the quality of a PRNG [3].

II. Background

A. True Random Number Generation versus Pseudo Random Number Generation

There are two different ways random numbers are currently produced in computing - true random number generation (TRNG) and PRNG. The TRNG generator produces numbers that are random in nature and have outputs that are unpredictable. The generator simulate sources of entropy that are random as in nature. Sources of entropy, for example, are thermal noise, noise from electrical signals, shot noise, atmospheric noise, radioactive decay or clock jitter [4]. So to produce a complete model of these natural sources of randomness is fairly straight forward and there are a number of ways to implement these types of RNGs. However, there are very limited applications of where these can be used which brings up the PRNG. The PRNG produces sequences of random numbers from a single seed value. The importance of this is that the PRNG is deterministic, meaning that using the same seed value in the same algorithm produces the same data set of random numbers [1]. In computer systems and particularly for experiments, this provides a better means of comparing outputs since the experiment is repeatable. Adjustments can then be made to compare to the same set of random numbers. It is also found that PRNGs can be typically more efficient overall, requiring fewer circuit elements, power, area and they can often produce random numbers quicker than TRNGs [1] [4].

B. Applications of PRNGs

These pseudo-random numbers are a powerful tool in development of a variety of different fields such as cryptography and Monte Carlo simulations [3]. In cryptography it can be useful to randomise pieces of information to serve as a means to protect data. In Monte Carlo simulations, large sets of random numbers are used as data to feed into the simulation. This is useful if one is trying to model a system that uses a random phenomenon to measure, recreate or re-purpose that system and/or its states. Referring back to cryptography, the information is encrypted by, to put it simply, randomising the information, and a key is used to decrypt the data after it has been transferred over a network. The encryption and decryption of the information requires that it be randomised in a predictable manner, thus the use of PRNGs [5].

For Monte Carlo simulations, very large amounts of random numbers need to be produced depending on the type of simulation. The Monte Carlo or random-sampling technique makes use of repetitive simulation of a subject of study, for example simulating conditions going from a point A to a point B such as adjusting weather conditions or terrain. This requires random numbers to be produced to adjust the variables in the simulation [6]. These Monte Carlo simulations are often executed repetitively depending on the simulation design and subject matter, between thousands and millions of times over. The benefit is that the faster these simulations execute, the less time-consuming the process can be - more simulations can provide more accurate results.

Random numbers are used in a wide range of different applications that may also include spread-spectrum communications [7], gaming, statistical analysis [8], modelling [9] and many more.

C. Factors Assessing Quality of Random Numbers

There are a wide variety of different aspects to assess the quality of a PRNG implementation, as there are many different implementations that may have more favourable results for one application, for example power consumption, execution speed and overall spread of the random numbers. Some typical performance measurement statistics include [10], [11]:

- The length of the period between repeating sequences of random numbers. Most kinds of PRNGs make use of an inputted seed value to calculate the first random number (the next state) which will then cascade repeatedly to produce the sequence of random numbers. However, the sequence is not infinite and at some point when a number already produced before in the sequence is reproduced again, the sequence begins to repeat. The Mersenne Twister, for example, has a maximum period of 2^{19937}-1.
- The 'randomness' in the random numbers, usually measured as a uniform probability distribution. A good spread in the random values means that there is no repeating of sequences or predictable behaviour of the RNG.
- Execution speed, complexity and cost to produce the algorithm used in PRNG. With a wide variety of custom PRNGs available and some well-known ones, there are often trade-offs in performance statistics of different algorithms. For example, the Mersenne Twister is considered to be a fast algorithm, not making use of multiplication and division, however it is very complex and costly to implement.

The correct algorithm should be chosen depending on the application's needs.

D. Serial versus Parallel Number Generation

The nature of random number generation is that large amounts of data need to be produced, and depending on the application, within a small amount of time. PRNGs work on a very sequential process, using an input seed to generate a random number, which is then repeated using a statistical or mathematical variation on the number to produce the next number or state. Some common PRNGs that make use of this process include the Linear Congruential Generators (LCGs), Mersenne Twister (MT), LFSRs and Multiple Recursive Generators, such as L'Ecuyer's MRG32k3a [12]. There is a drive to develop parallel implementations of algorithms such as the ones mentioned, however doing so increases the overall complexity of the algorithm and potentially ruins the overall random number generation quality. There are a number of techniques that are commonly used in parallelisation of these algorithms, including leapfrogging (Figure 2), sequence splitting (Figure 3) [12] and random-tree (Figure 1). The following figures illustrate these concepts [13].

In each element of the diagrams, random number generation is split amongst separate processors. The random tree implementation replicates the same process or algorithm but produces multiple outputs of the same value on each iteration. The leapfrog technique takes the same algorithm, splits it among separate independent processing elements and generates the random numbers. Each processor produces

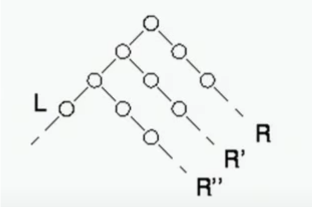

Fig. 1: Random tree technique for parallelisation

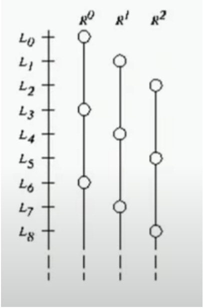

Fig. 2: Leapfrog technique for parallelisation

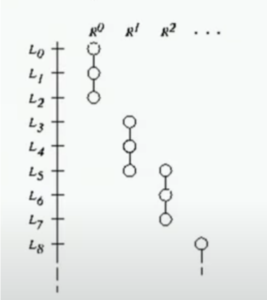

Fig. 3: Sequence splitting technique for parallelisation

a random number in a specific location in the sequence. Sequence splitting is similar to leapfrogging, however multiple numbers are produced sequentially at a time. Once these techniques are completed, the sequence can be reconstructed by combining the processor outputs. The issue with these parallelisation techniques is that there needs to be some way to predict the next number or state in the sequence, or better yet any number in the sequence, which drastically increases complexity. The mathematics behind these algorithms is non-trivial to say the least.

E. Linear Feedback Shift Register RNG

The LFSR is a common type of RNG that uses shift properties to randomise bits within a number. The LFSR makes use of exclusive-or (XOR) gates and shift registers, which makes this method suitably realisable on hardware based architectures, such as an FPGA [2]. The operation of the LFSR calculates the next state, according to Figure 4 and is described, in detail, below.

With reference to Figure 4, if the value of the least significant bit (LSB) is one, then the LFSR will perform a shift right where the LSB becomes the most significant bit (MSB) and effectively the bits in the location described by the feedback polynomial (see equation 2) will be inverted. If the LSB is zero, then the operation is performed as described previously, however it does not perform the inversion. There are two versions of the LFSR, the Fibonacci and the Galois implementations [14]. Both are very similar except that the Galois can perform multiple XORs at once, making it better for parallel applications [14].

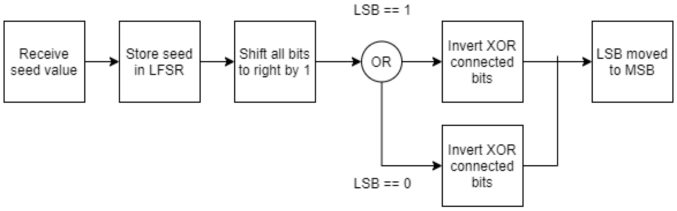
Receive seed value
Store seed in LFSR
Shift all bits to right by 1
LSB == 1
OR
Invert XOR connected bits
LSB moved to MSB
LSB == 0
Invert XOR connected bits

Fig. 4: LFSR operation block diagram

The major element of the LFSR is that it produces a large period between repeating sequences of numbers, with good uniform distribution of the random number numbers over its sequence. The period is determined by the size, more specifically the number of bits, of the input number it is designed to accept. To maximise the period, XOR gates are placed at specific locations in the LFSR and are defined by what as known as the maximum feedback polynomial. The following example is a maximum feedback polynomial for a 16-bit LFSR [2].

$$X^{16} + X^{14} + X^{13} + X^{11} + 1 \tag{1}$$

The exponents define the locations of the XOR gates that are used in the feedback of the LFSR. An illustration of this can be seen in Section IV, Figure 6 for a 32-bit feedback polynomial.

III. Methodology

A. Hardware

Before the design can be implemented, it was important to note the available hardware. The computer used to run the C++ code as well as the Verilog code is a laptop computer consisting of the following specifications:

- Intel®Core™ i7-6500 @ 2.50GHz quad core CPU
- Windows 64-bit operating system
- 8GB RAM with 7.71 GB usable memory
- 250GB HDD with 231 GB usable storage

The FGPA that would be used to run the parallel implementation is the Nexys A-7 Artix 100t board from Digilent®with the following specs.

- 15,850 logic slices, each with four 6-input LUTs and 8 flip-flops.
- 4,860 Kbits of fast block RAM.
- 6 clock management tiles, each with phase-locked loop (PLL)
- Internal clock speeds exceeding 450MHz
- 128 MiB DDR2 Serial Flash

B. Experimental Procedure

The design cycle of the project followed a format based on the pattern provided in Figure 5.

The design process was based on a waterfall model, with a fall back onto redesigning and prototyping to accommodate for unforeseen design flaws.

From analysis of the project description, the following specifications were drawn:

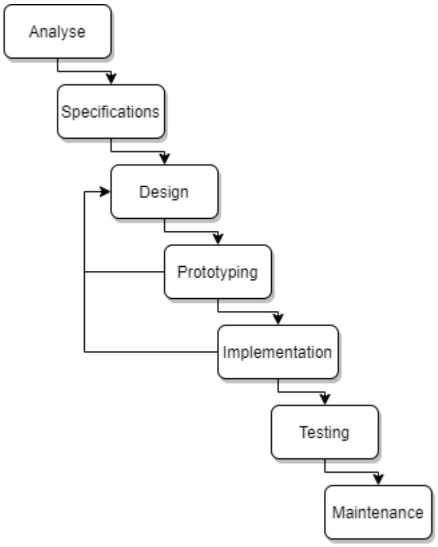

Fig. 5: Design cycle of the project

- The PPRNG should produce N random numbers in one go.
- The random numbers produced should range between 0 to 2^{32}-1.
- The random numbers need to be stored in RAM, which can be accessible to a soft-core processor.
- Inputs to the system are: unsigned seed, unsigned start address, unsigned count (i.e. the number of random numbers to generate), bit activate.
- Outputs of the system are: ”bit busy” and the random numbers stored in RAM.

The next step was to determine the RNG to use and adapt to a parallel implementation. The factors that influenced the decision-making process were based on the following factors: hardware realisibility, simplicity, large period between repeating sequences, low hardware resource consumption and ability to create a parallel version. It was important to be hardware realisable so that it could be implemented on the FPGA in a straight forward manner, as it was hypothesized that increased complexity on hardware architectures can increase power consumption, hardware resources and difficulty in producing the alogrithm.

Once a preferred algorithm was selected, the serial implementation was then produced and established as a Golden Measure.

The implementation was then produced based on the Golden Measure in Verilog. Once a working implementation was created, the parallel aspects could then be designed, which needed to incorporate the specifications detailed at the start of the project development. This stage required an iterative stage of redesigning of the project implementation, including the Golden Measure.

Finally, once suitable outputs were being produced, the system could then be tested. This includes analysing execution speeds, recording speedup or slowdown as well as performing some basic statistical tests on the output of the captured data between the two implementations. The execution speeds are used to determine the speed-up between the implementations, so that a comparison between a serial and parallel implementation can be made. The results of the experimentation will, potentially, support the use of this designed parallel implementation.

In terms of measuring the speed of the implementations, this was measured using libraries provided in C^{++} to record the wall clock time taken to execute the algorithm, while for Verilog the simulation provides the estimated execution times and these were sufficient for comparisons.

The final steps were to perform maintenance and final adjustments of the project to finalise the product design.

IV. Design

A. 32-bit LFSR Design

The design of the LFSR makes use of shift registers and exclusive-OR (XOR) gates to randomise an inputted seed value. The input seed is stored in the shift registers, where specifically selected bit positions in the seed are fed to an XOR gate. The positions of these XOR gates are critical in developing the maximum sequence of random numbers that can be generated by this seed, where these positions are defined by what is known as the feedback polynomial [2]. The maximum period achievable by this LFSR design is 2^{32}-1, meaning that 4 294 967 295 numbers can be produced before the sequence repeats. The maximum feedback polynomial to achieve this with a 32-bit number is defined in Equation 2 below.

$$X^{32} + X^{22} + X^{2} + X^{1} + 1 \tag{2}$$

Where 32, 22, 2 and 1 define the locations of the XOR gates. The following block diagram in Figure 6 illustrates the concept.

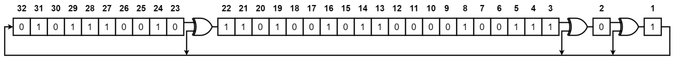

Fig. 6: 32-bit LFSR design using shift registers and XOR according to the feedback polynomial

This design for the LFSR is then emulated on the serial, C^{++} implementation for the Golden Measure and then adapted for a parallel implementation, where multiple LFSR units are created and run.

B. Serial implementation (Golden measure)

The implementation provided is based on the Galois LFSR [14] but has been adapted for the parallel seeding (see Section IV-C for details on this). To produce the program, *Code::Blocks* was used along with the GCC Compiler for Windows 10.

The code segment provided in Listing 1. provides the code used in generating the random numbers.

```
unsigned lfsr_galois(void)
do
    {
        //Output current seed based on period
        separate_seed = start_seed + 314159265*period;

        lfsr = separate_seed;
        unsigned lsb = lfsr & 1;    /* Get LSB (i.e., the output bit). */

        lfsr >>= 1;                 /* Shift register */
        if (lsb)                    /* If the output bit is 1, */
            lfsr ^= 0x80200003;     /* apply toggle mask:
            0b10000000001000000000000000000011 */
                                    /* this relates the 32-bit Maximum Feedback
                                    polynomial.*/
                                    /* ^ is an XOR operation. */

        for (int i = 0; i < 32; i++)    //for displaying the number as a string
          {
             s[31 - i] = (lfsr & (1 << i)) ? '1' : '0';
          }
        printf("%u\n", lfsr);

        //Writing to memory
        memory[period] = lfsr;
        period++;
    }

    while (period != numbers);
    {
        gettimeofday(&end, 0);
        long seconds = end.tv_sec - begin.tv_sec;
        long microseconds = end.tv_usec - begin.tv_usec;
        printf("Time measured: %.3f microseconds.\n", seconds*1e6 + microseconds);
        //
        return period;
    }
```

Listing 1. Serial implementation of the LFSR to produce random numbers sequentially

The code is inherently sequential, where each number or state is produced after the previous number has been produced. The adjustment made to the algorithm was the incorporation of parallel seeding. For each iteration the initial seed that is inputted to the system is altered by adding a multiple of a number that was arbitrarily chosen. This was used to produce the parallel implementation, where more details for the reasoning of this design choice is described in Section IV-C. For an understanding of how the code works, it uses the same process detailed in Figure 4 to randomise the input seed.

The output of this code for four random numbers with an initial seed value of 3429426846, produced the following:

```
1714713423
4021373852
2028872688
2188049475
Time measured: 998.000 microseconds.
```

The time is measured using the sys/time.h library to provide the time it takes to execute the program in real time. To ensure minimal overhead in recording the time, it is recorded at the start of the function that generates the random numbers and ends before the function returns a value, signifying it has ended.

The only change that will be implemented in the code during testing is the aspect of writing to memory. For this implementation, memory is just defined to be an array to which the random numbers are written. This will be removed in tests to compare speeds of the different implementations without memory to examine the performance only of the algorithm itself, or kept in for performance comparison including the time to write to memory. Thus if a host program was to make use of this implementation, it would just need to be able to access that array address to read values from it.

C. Parallel Implementation (Verilog Measure)

The parallel measure was broken down into smaller building blocks which could later be combined to hopefully produce the desired result.

The first building block consisted of producing a serial LFSR in Verilog where an implementation was altered from [15] to produce a version that would work for the designed scenario. The LFSR Verilog code can be found in Listing 2. IV-C, this code was combined with other building blocks to produce the parallel implementation.

An important step in designing the parallel implementation of the LFSR in Verilog included the use of 'For-loops'. 'For-loops' do not behave in the same fashion as those in regular programming as they simply replicate the code **i** times instead of looping through the code **i** times at synthesis time. This proved an efficient way to produce multiple LFSRs that operated all together at the positive edge of the clock. It was found through debugging an error that a maximum of fifteen LFSRs could be run per clock cycle, thus limiting the maximum loop iteration to fifteen, which also means fifteen random numbers per clock cycle. Therefore, the design specifications need to change as not all the numbers can be produced at the same time.

In between developing the serial and parallel versions of the RNG an issue was encountered with regards to the seeding of the parallel version of the RNG. The issue being to generate the same random number set as the serial implementation. Due to the relationship between consecutive random numbers in the serial implementation, the random number from the previous LFSR operation is used as the seed for the next random number. For the parallel version to produce the same random number set as that of the serial measure, it would require a predictive method capable of producing **n** required seeds for the individual LFSRs all at once. This described operation was felt to be out of the scope of the project and adaptive measures were taken to alter the algorithm.

The adapted method was to increment each seed in a predictive manner. Using the 'For-loop', the current iteration index of the 'For-loop' i.e. (int i) will be used to increment the seed thus providing each LFSR with a unique seed.

A concern from this previous seed incrimination method was the quality of the randomness of the numbers produced. This concern was realised as the variation in numbers produced was poor since consecutive numbers only varied by a few units.

In an attempt to resolve this, the iteration index of the 'For-loop' was multiplied by different sized numbers. The number was chosen arbitrarily through an iterative process where the randomness of the resulting random numbers was considered due to the chosen multiplier. The multiplier ranged from 1 to 9-digit numbers, it was stopped at 9-digits as Vivado limited it as such. Multiple 9-digit numbers were

```
shiftVal = seed + checkN*314159265;     // Make each seed unique
shiftVal = {shiftVal[0],shiftVal[31:1]};
//concatenate lsb to msb - essentially an arithmetic shift right
if (shiftVal[31])begin  //check value of lsb of seed
shiftVal = shiftVal ^ 32'b00000000001000000000000000000011;
                //apply bitmask (based on optimal values to XOR of 32 bit number)
                //essentially XOR the values at bit-positions 1, 2 and 22
```

Listing 2. Snippet of the LFSR Verilog implementation showing the coded process to produce a psuedo random number with an LFSR

```
if((countB == n))begin //When A and B address at their max, i.e Done
    r_busy <= 0;
    r_readReady <=1;   //ready to read
    countA <= 10'b0;   //reset counts for read operation
    countB <= (n>>1);
end
else if(!r_busy && r_prngDone)begin
    countA <= 10'b0;  //Port A starts at the lowest address
    countB <= (n>>1); //Port B starts at half way from the maximum
    r_busy <= 1;      //Busy sending data to memory.
end
else begin
    addra <= countA + startAddr; //Move first half of array to BRAM + start address
    addrb <= countB + startAddr; //Move second half of array to BRAM + start address
    dina <= array[countA];       //Add data at the current A address to dina
    dinb <= array[countB];       //Add data at the current B address to dinb
    countA <=countA +1;          //Increment A address
    countB <=countB +1;          //Increment B address
end
```

Listing 3. Verilog logic for writing to BRAM given a start address and **n** random numbers, this is located in an always block and is clocked on the positive clock edge when it is needed.

tested, however, it was settled on the first 9 digits of π as the results showed a reasonable standard of randomness from initial observations.

The Random Access Memory (RAM) aspect of the project was implemented through Block RAM (BRAM) IP core functionalities located on the FPGA device. The different settings of the BRAM IP were looked up in the datasheet to decide on which functionalities would suit the application best. The BRAM was chosen to be set-up in ”True-Dual Port Fashion” allowing simultaneous write and read to the allocated BRAM through two ports thus halving the write and read time. The code to write to memory with a given starting address is given in Listing 3 IV-C. The maximum amount of random numbers that could be stored on the FPGA was calculated to be as follows with a given BRAM memory size of 64kB:

$$\frac{64 \times 10^3 \times 8\, bits}{32\, bits} = 16000\, random\, numbers. \tag{3}$$

The aforementioned components of the parallel design were integrated together to produce the final design of the Verilog implementation. The block diagram in Figure 7 shows the interfacing between the *top module* and BRAM as well as their respective inputs and outputs. There is also ”hand-shaking” (acknowledgment between communicating modules in an integrated computer structure) shown with certain registers associated with inputs to the BRAM module.

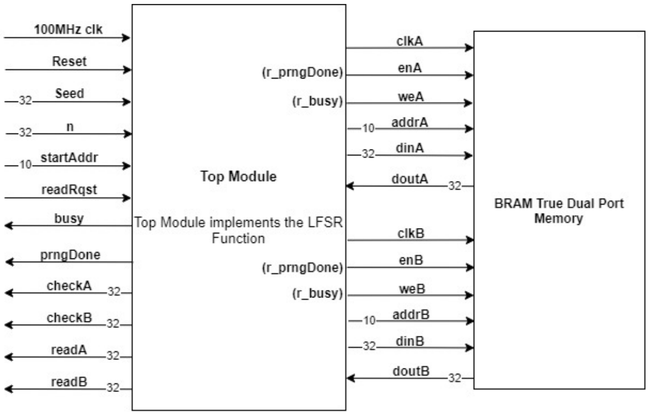

Fig. 7: Verilog Block diagram showing interfacing between top module and BRAM final design

V. Proposed Development Strategy

The main advantage of using PPRNGs over sequential predecessors is the inherent speedup. The increase in computing power and availability of larger amounts of resources on smaller area chips is providing the ability to make use of parallel computing techniques to complete tasks. Particular applications that would benefit greatly from this speedup are Monte Carlo simulations. Monte Carlo simulation is a tool used in a variety of different applications such as probability simulation and forecasting models [16]. These simulations can provide a large amount of information relating to the system it is applied to, for example risk analysis and prediction. It works on the principle of when the system is modelled, it can be executed with variations thousands of times over to provide very accurate results in order to draw trustworthy predictions [16]. Having a digital accelerator that is capable of delivering random numbers that are fed into the simulation at greater speeds due to the reduced latency in producing those numbers greatly improves the rate at which these Monte Carlo simulations can be run, allowing for more executions in a given time period for increased number of results. The amount of random numbers varies according to the application, but sources often quote values higher than 10^{10} and sometime values even larger than that [17].

For the purposes of this project, the FPGA can suitably produce large amounts of random numbers to be transmitted to the application making use of the Monte Carlo simulation. With a few design alterations, such as including more RAM to store more random numbers, the FPGA could provide a suitable means to generate random numbers in a small amount of time, so long as the reading from the BRAM does not cause significant overhead. The dual port nature of the BRAM can also facilitate parallel reading of numbers to the application. The next step in the project is to expand on the ability of the PPRNG developed to utilise more of the FPGA that is available, if the application it is applied to does not require excessive amounts of the FPGA's resources for its own processing.

There are additional fields that could be explored to utilise the product developed. One particularly notable field is the use of LFSR in fault grading of Application Specific Integrated Circuit (ASIC) design. Designers and engineers can use these for a high level fault coverage of ASIC designs with minimal effort to test the circuitry [18]. This works by connecting the bits of the LFSR to inputs of the ASICs, where the LFSR can then be run repeatedly to examine outputs of ASIC circuits [18]. Due to the very large sequence of random numbers produced by the LFSR, it produces all the necessary combinations of inputs to examine if all the outputs produce the correct results. For this product, the parallel implementation would be able to provide more connections for ASIC testing. The numbers stored in BRAM could be read to identify the pattern of the bits used to test specific inputs of the ASICs for the error analysis. The limitation in the current design would be the input-output (I/O) connections of the FPGA, which are limited especially for the NEXYS A7 Artix 100-t, but the ability for additional LFSRs produced on a single device for testing of ASIC designs presents a considerable area of application for this product.

VI. Planned Experimentation

This section will detail how the results will be obtained for the project. There were two separate programs coded, as mentioned previously, a golden standard coded in C^{++} and an equivalent parallel version coded in Verilog. The code will be tested and compared with the following experimentation.

All results will be tabulated and graphed. They are all shown in the results section (see Section VII below).

A. Golden Standard

The C^{++} program was coded with a wallclock method for timing as seen in Listing 1 IV-B to allow random numbers to be generated along with the time taken to generate that sequence on the processor. The timing will be calculated as the time it takes to generate the random numbers and write them to memory (memory being a pre-sized array). The code will be tested to extract the following two relationships in the random data: firstly the relationship between the varying seed and the time taken to generate random numbers and secondly to find the relationship between the time taken to generate random numbers versus the quantity of numbers being generated. These two tests will be run concurrently by increasing the quantity of numbers generated for one seed and then varying the seed and running the same test of increasing the quantity of numbers to generate for each seed. There will be five different seeds that will be tested and these are shown in list VI-A.

1) 123456
2) 654321
3) 1
4) 3429426846
5) 6386547

These seeds were randomly chosen with the following objectives in mind:

- Test 1-bit sized seed
- Test 32-bit sized seed
- Test seed with unequal bits
- Test seed with incrementing bits
- Test seed with decrementing bits
- Test arbitrary seed

The tests will be run ten times and then averaged out to give a more balanced set of results.

The quantity of numbers to generate with the number generator is shown in list VI-A. The number will be increased in powers of 10 starting at 1 until the maximum size of the BRAM is reached. The maximum number of 32-bit numbers which can be stored was calculated as 16 000 (refer to equation IV-C) and thus this will be the maximum number of random numbers, which will be generated.

- 1
- 10
- 100
- 1 000
- 10 000
- 16 000

B. Parallel Program

The parallel program was coded to be identical in outputs to the golden standard (see section IV on code implementation), therefore the exact same set of tests will be conducted on the Verilog program as was performed on the C program. The Verilog code is all based on clock cycles in simulation and thus it is assumed that the seed will not impact time and as such only one seed will need to be tested. Two of the seeds will be tested to confirm/contradict this assumption. The Verilog code will be tested, as previously mentioned but will also incorporate timing of generating the random numbers as well as timing the results being written to memory - where the C^{++} code was only timed after writing to memory. This will be done to give an idea of how long the program takes for number generation versus writing to BRAM.

C. Comparison

To compare the results from the two programs, the speedup for results will be calculated using equation 4. Before speedup is calculated, the mean value of all the speeds of each seeded test will be calculated with respect to N. This average will then be used to calculate speedup of the C^{++} Golden Standard over the parallel Verilog implementation.

$$speedup = \frac{T_{p1}}{T_{p2}} \tag{4}$$

D. Statistical Analysis

The last test to be done is to verify the entropy of the data produced by the random number generator i.e. how 'random' the numbers are. This will be done by producing 1000 numbers with the Golden Standard with an arbitrary input seed (5462994) and then producing another set of 1000 numbers with the original galois Serial LFSR number generator in C^{++}. These two sets of results will be graphed and compared to analyse the entropy of the LFSR parallel implementation.

VII. Results

The results for the tests that we performed on both the Golden Measure and Parallel implementation are shown and discussed in the following section.

A. C^{++} Golden Standard

The tests discussed in Section VI were performed and are shown in table I. There is a clear increase in the speed taken to generate numbers with the increase seeming quite linear i.e. 10x increase in quantity relates to a 10x increase in time taken but this can be seen better in figure 8 where the linear relationship between the quantity of random numbers generated and the time taken can be seen up until ten thousand numbers generated where the time starts to stabilise and the relationship is no longer linear. It is also obvious from the graph that the seed has little impact on the time taken and all seeds produce a similar result in time. The input seed 1, however, produces the largest variation in speed clearly taking longer than the other input seeds, which all have very similar times.

TABLE I: Table for the average time taken to generate increasing quantities of random numbers with varying input seeds for the C++ implementation of the LFSR random number generator

	Time (s) with specific input seed				
Quantity of Random Numbers Generated (N)	**1**	**3429426846**	**123456**	**654321**	**6386547**
1	9.960E-5	9.960E-5	2.991E-4	9.720E-5	9.920E-5
10	9.775E-4	1.194E-3	1.590E-3	1.792E-3	1.687E-3
100	5.934E-2	6.444E-2	7.046E-2	7.046E-2	6.971E-2
1000	7.770E-1	7.802E-1	7.242E-1	7.388E-1	7.688E-1
10000	6.823E0	6.654E0	6.201E0	6.432E0	6.662E0
16000	7.613E0	7.074E0	6.919E0	7.142E0	6.857E0

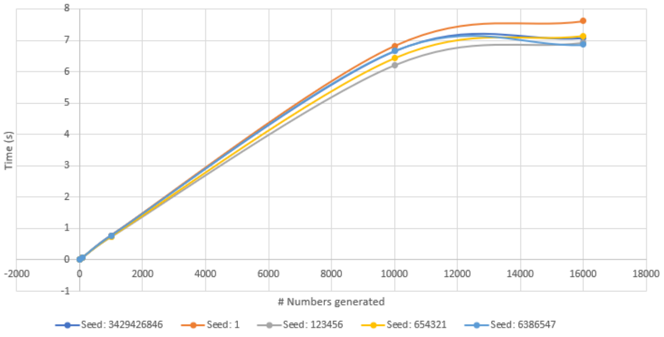

Fig. 8: Graph showing the relationship between time taken versus quantities of random numbers produced for different input seeds for the C^{++} implementation

B. Verilog Parallel Implementation

The results of testing the Verilog program are tabulated in II showing a linear increase in the time taken to generate random numbers as N is increased. This result is once again best seen on the graph shown in figure 9 where there is a directly proportional relationship of the time taken to produce N numbers. The other important result to note is that as N increases, the time taken to store the numbers in BRAM increases drastically as would, intuitively, be expected.

The input seed, as expected, made no difference on the time and as such only one seed test was run and is shown below.

It was important to note that during the test, the Verilog implementation was unable to produce more than 15 numbers in a single clock cycle.

TABLE II: Table showing the average time taken to generate increasing quantities of random numbers with two program variations. One variation was generating the numbers and printing them to the output console, the other variation generated the numbers and wrote them to BRAM. This is for the Verilog implementation of the LFSR random number generator

	Time (s) for generating random numbers and writing to memory	
Quantity of Random Numbers Generated (N)	Generating Numbers	Generating and writing to BRAM
1	1.00E-8	3.00E-8
10	1.00E-8	7.00E-8
100	7.00E-8	5.80E-7
1 000	6.70E-7	5.68E-6
10 000	6.67E-6	5.67E-5
16 000	1.07E-5	9.07E-5

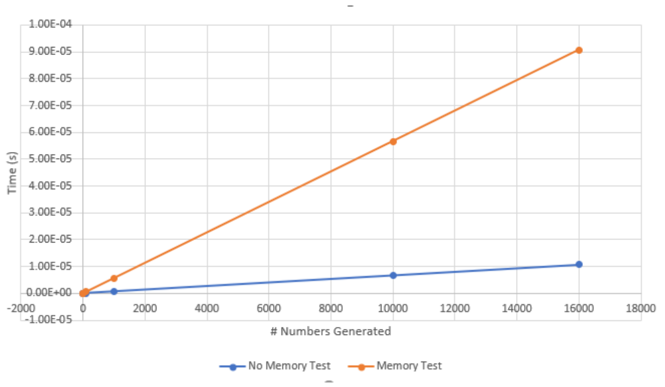

Fig. 9: Graph showing the relationship between time taken versus quantities of random numbers produced for generating random numbers and writing to BRAM for the Verilog program

C. Serial versus Parallel

The table showing the speedup of the serial program over the parallel program by using equation 4 with respect to N is shown in table III. The results show that as N increases the speedup increases to a very large degree up until 10 000 numbers where it still increases but by a smaller factor than before. The relationship can be seen in figure 10 where there is a huge increase seen up until 10 000 numbers produced where the speedup begins to plateau and then decrease at 16 000 numbers. There is still a significant speedup observed even if it is not as large as the smaller data-sets.

TABLE III: The speedup of the Golden Measure over the Parallel implementation as N is increased. This metric includes the time taken to write to memory for both programs.

Quantity of Random Numbers Generated (N)	**Speedup**
1	4.63E3
10	2.07E4
100	1.15E5
1 000	1.33E5
10 000	1.16E5
16 000	7.85E4

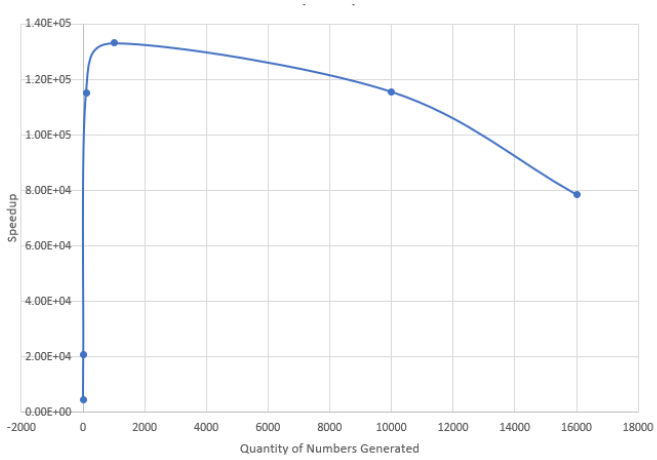

Fig. 10: Graph showing the speedup of the Golden Standard over the Parallel program as N is increased

D. Statistical Analysis

The data-sets of two separately generated random number generators are graphed in figure 11. It can be noted that the Galois LFSR produces far better results as there is no pattern noticeable in the graph - it closely resembles a random noise process. The bottom graph however, even while showing different numbers that are considered 'random', shows a clearly predictable pattern on the graph.

The numbers generated in both the serial and parallel implementations, were all unsigned decimals (non-negative) and were no larger than 4 294 967 295. The maximum number that was generated across all the tests performed ended up being 4 293 237 909, though this is not to say that the algorithm is not capable of reaching the maximum value but rather that the tests performed did not generate enough numbers to produce a single result that was the maximum.

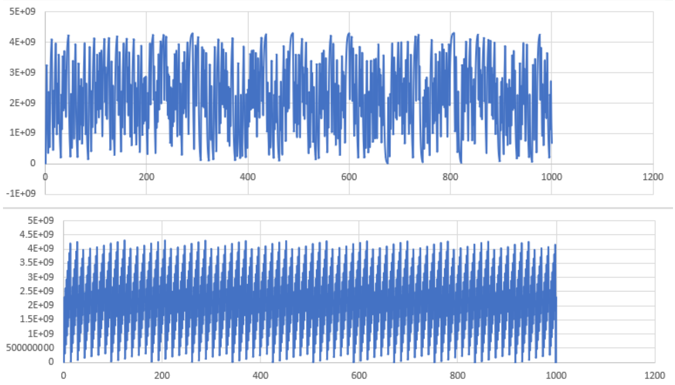

Fig. 11: Two graphs showing 1000 seperately generated random number sets. The top graph is the Galois serial LFSR and the bottom graph is the edited LFSR that was used as a golden measure against the verilog parallel implementation

E. Discussion

The results yielded from testing the digital accelerated version of an LFSR random number generator implemented in parallel showed three key points:

- The parallel version of the RNG is far more efficient in terms of producing random numbers - yielding a speedup of up to 133 000 when 1000 numbers were produced and even though the speedup was not as large at the maximum storage value, a speedup of 78 540 was achieved when 16 000 numbers were produced. The FPGA is optimised for parallel programming [19] and this can be directly seen with the vast speedup results achieved with the Verilog implementation. The processor on the PC has to wait for the (n-1)th number to be produced in the serial version before it can produce the nth number. The speed is, as a result, very slow in the golden standard especially for large number sets where it takes several seconds to produce the data-set when the parallel version takes less than a millisecond.
- The seed has little to no impact on the speed of the program in both implementations, especially in the digitally accelerated parallel program, where it has absolutely no impact on program speed. The LFSR works with bit operations. It checks the value of the LSB after shiting it. The LFSR is therefore performing the same operation regardless of the value of the bit (the bit has value 1 or 0). As a result, the seed should not impact the speed of the RNG. The results agreed with this hypothesis.
- The LFSR algorithm used to produce the sets of random numbers, although consistent in producing random numbers and producing the exact same sets of numbers for the serial and parallel programs, does not produce a good entropy i.e. the randomness of the sets of numbers. This is because there is a slight pattern to the numbers produced even if it is not immediately visible when observing the numbers. The Galois LFSR has limits in its applications because of the fact that it is predictable. The method is the same for all applications unless the feedback polynomial has been changed, which affects the statistical quality of the output of the LFSR. Therefore, it is safe to assume that very few applications would change this. The variation made to the LFSR in order to parallelize the program made this even more predictable in its patterns. For the purpose of producing random sets of numbers, this random number generator is sufficient. In terms of applying this generator to real-life programming applications, this LFSR iteration is quite limited.

The LFSR could be improved by finding a better way to randomise the input seeds - this is fundamentally the issue with the LFSR implemented throughout this experiment and ultimately one of the biggest concerns facing parallel random number generators - how does one randomise the seeds to get multiple predictive and repeatable input seeds from one seed? One of the ways to get around this problem - instead of randomising the seed one can implement a form of ”Leap-frogging” to the parallel LFSR implementation. The maths behind this is complex and difficult to implement in programming but would be one of the ways to go about fixing the entropy of the generated data-sets. If this was achieved, the results would resemble the Galois LFSR results shown in 11. The parallelisation in every other aspect can be considered a success because of the fact that the parallel version produced repeatable and very large data sets of numbers, which are sufficiently random, in a parallel manner and in a hugely accelerated version when compared to the golden standard.

The specification for the parallel version of the LFSR PPNG was changed to accommodate the fact that only fifteen random numbers could be produced per clock cycle when the original specification was to produce all numbers "at once". This limitation is potentially a result of the Vivado simulation and further testing would need to be performed to fix this fault if more than fifteen random numbers should be produced in one clock cycle. This fault was however acceptable since the parallel version achieved huge speedups over the golden standard even with the limitation of fifteen numbers per clock cycle.

VIII. Conclusion

An LFSR serial random number generator was successfully implemented in a C^{++} Golden Standard and run on a laptop processor. This same algorithim was then applied and implemented through Verilog code to be run on an FPGA board. The digitally accelerated version of the random number generator yielded large speedups up to 78 540 for the largest data-set. The parallel code met all specifications with successful implementations of address, count and bit activate inputs. The only shortcomings of the parallelised code were the predictability or lack of entropy in the data-sets produced as well as that the PPRNG could not produce more than fifteen numbers in one go. The original Galois LFSR C^{++} serial program gave far better results than the edited parallel version in terms of predictability and randomness. This drawback highly impacts the usefulness of this program, limiting it to be mostly used for Monte Carlo simulations or fault checking in ASIC design rather than cryptography. The digital accelerator in the form of a parallel random number generator did however yield acceptable results with the data-sets being, practically considered, 'random'.

Link to GitHub:Verilog and C++ Code Repository

References

[1] "Pseudo random number generator (prng)," Sep 2019. [Online]. Available: https://www.geeksforgeeks.org/pseudo-random-number-generator-prng/

[2] A. K. Panda, P. Rajput, and B. Shukla, "FPGA Implementation of 8, 16 and 32 Bit LFSR with Maximum Length Feedback Polynomial using VHDL," in *IEEE International Conference on Communication Systems and Network*, vol. 2. IEEE, May 2012, pp. 769 – 773.

[3] P. L'Ecuyer, *Handbooks in Operations Research and Management Science: Simulation, Chapter 3 Uniform Random Number Generation*, 1st ed. Elsevier Science, 2006, vol. 13, p. 55–81.

[4] A. Yadav, "Design and analysis of digital true random number generator," 2013. [Online]. Available: https://core.ac.uk/download/pdf/51290139.pdf

[5] R. Soorat, K. Madhuri, and A. Vudayagiri, "Random number generator for cryptography," *Nanosystems: Physics, Chemistry, Mathematics*, p. 600–605, Oct 2017.

[6] "Monte carlo methods in practice," Apr 2015. [Online]. Available: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice

[7] R. Pickholtz, D. Schilling, and L. Milstein, "Theory of spread-spectrum communications - a tutorial," *IEEE Transactions on Communications*, vol. 30, no. 5, p. 855–884, 1982.

[8] M. Nazarathy, S. Newton, R. Giffard, D. Moberly, F. Sischka, W. Trutna, and S. Foster, "Real-time long range complementary correlation optical time domain reflectometer," *Journal of Lightwave Technology*, vol. 7, no. 1, p. 24–38, 1989.

[9] G. Jandaghi, A. Gaeini, and A. Mirghadri, "A general evaluation pattern for pseudo random number generators," *Trends in Applied Sciences Research*, vol. 10, no. 5, p. 231–244, 2015.

[10] F. James and L. Moneta, "Review of high-quality random number generators," *Computing and Software for Big Science*, vol. 4, no. 1, 2020.

[11] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, p. 3–30, 1998.

[12] T. Bradley, J. D. Toit, R. Tong, M. Giles, and P. Woodhams, "Parallelization techniques for random number generators," *GPU Computing Gems Emerald Edition*, p. 231–246, 2011.

[13] *Math for Game Programmers: Parallel Random Number Generation*. Game Developer's Conference, May 2018. [Online]. Available: https://www.youtube.com/watch?v=qO8FAQvlgX0

[14] "Linear-feedback shift register," Jun 2020. [Online]. Available: Linear-feedbackshiftregister

[15] NANDLand, "Lfsr in an fpga - vhdl amp; verilog code." [Online]. Available: https://www.nandland.com/vhdl/modules/lfsr-linear-feedback-shift-register.html

[16] "What is monte carlo simulation?" [Online]. Available: https://www.riskamp.com/files/RiskAMP%20-%20Monte%20Carlo%20Simulation.pdf

[17] H. Bauke and S. Mertens, "Random numbers for large-scale distributed monte carlo simulations," *Physical Review E*, vol. 75, no. 6, Apr 2007.

[18] *What's an LFSR?* Texas Instruments, 1996. [Online]. Available: https://www.ti.com/lit/an/scta036a/scta036a.pdf

[19] D. C. Marinescu, "Parallel hardware," 2013. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/parallel-hardware

Fig. 12: pew pew