12-2006

A Reconfigurable Supercomputing Library for Accelerated Parallel Lagged-Fibonacci Pseudorandom Number Generation

Yu Bi
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Yu Bi entitled "A Reconfigurable Supercomputing Library for Accelerated Parallel Lagged-Fibonacci Pseudorandom Number Generation." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Donald W. Bouldin, Robert J. Harrison

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Yu Bi entitled "A Reconfigurable Supercomputing Library for Accelerated Parallel Lagged-Fibonacci Pseudorandom Number Generation." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Gregory D.Peterson
Major Professor

We have read this thesis
and recommend its acceptance:

Donald W. Bouldin

Robert J. Harrison

Accepted for the Council:
Linda Painter
Interim Dean of Graduate Studies

(Original signatures are on file with official student records.)

A Reconfigurable Supercomputing Library for Accelerated Parallel Lagged-Fibonacci Pseudorandom Number Generation

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Yu Bi
December 2006

ACKNOWLEGEMENTS

First, I would like to thank my advisor, Dr. Gregory D. Peterson for his constant support and instructive guidance. Second, I would like to thank Dr. Robert Harrison for his support and great suggestions of my research and study. Third, I would like to thank Dr. G.Lee Warren for his insight, explanation and instruction during my work. I would also like to thank Dr. Don Bouldin for introducing me to FPGA design and serving on my committee member.

A special thank also goes to my colleagues including Junqing Sun, Akila Gothanaraman, Saumil Merchant, Shaoyu Liu, Zhenzhen Liu and Scott E. Fields. Without their help, this work would not have been possible. To my friends, I would also like to acknowledge them for their constant help and the happiness that they brought during my study life.

This work was partially supported by the University of Tennessee Computational Science Initiative and the National Science Foundation grant CHE – 0625598.

I dedicate this thesis to my mother and father.

ABSTRACT

To help promote more widespread adoption of hardware acceleration in parallel scientific computing, we present portable, flexible design components for pseudorandom number generation. Due to the success of the Scalable Parallel Random Number Generators (SPRNG) software library in stochastic computations (e.g., Monte Carlo simulations), we developed an efficient and portable hardware architecture fully compatible with SPRNG's Parallel Additive Lagged Fibonacci Generator (PALFG). Our general design produces identical results for all the parameter sets that SPRNG supports and yields high performance parallel random number generators which can each generate 162 million 31-bit uniform random integers per second on Xilinx Virtex II Pro FPGAs. The friendly design interface makes it easy for users to integrate into their applications, particularly computational scientists unfamiliar with reconfigurable hardware. Due to its fast generation speed and friendly interface, this uniform random number generator is being targeted as an open core for parallel scientific computing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

In many computational science and engineering applications, performance acceleration is the most essential and crucial issue. Moreover, the accuracy of stochastic computation results is critically influenced by the quality of the random number generators [1]. This thesis addresses both these issues by developing a hardware accelerated implementation of a good random number generator for parallel scientific application to accelerate the performance.

This introduction examines the need for fast random number generators to accelerate scientific and engineering computations. Chapter 2 presents a study of relevant previous work. Chapter 3 gives an overview of the design process and algorithms. Chapter 4 presents the design implementation, and Chapter 5 presents a discussion of the results. Chapter 6 addresses the implementation application on the Cray XD1 supercomputer. Finally, Chapter 7 summarizes, gives conclusions, and presents possibilities for future work.

1.1 The Growing Need For Good Random Number Generators

Random number generators are used intensively in many areas. One important application is cryptography. With the expanding use of digital communications such as computer networks, the Internet, and wireless communication, the need for the protection of transmitted information using cryptography is greatly growing. Random numbers are used to generate cryptographic keys, initialization vectors, padding bits and blinding values [2]. To ensure the security, good random number generators with good statistical properties are required. For example, in data security applications, the weakness of random number generators could be used to perform cryptographic attacks. Thus, good random number generators play a key role in this the strong cryptographer applications.

Another important area that needs random number generators is Monte Carlo simulation. Monte Carlo simulation is a method of simulating a physical process using random numbers to sample the state space. With the tremendous advances in serial and parallel computing, the Monte Carlo method has evolved to tackle many complex problems in such diverse areas as nuclear medicine [3], finance [4], and computational chemistry [5]. Being statistical techniques, MC methods often require a large number of random samples to reduce the statistical error in a computed result. Therefore a good random number generator with long period and minimum correlation is desirable in MC applications. Without good random number generators, unwanted statistical bias and incomplete sampling will appear, which will result in the questionable result from the MC simulations.

1.2 Reconfigurable Computing

Reconfigurable computing (RC), the use of programmable logic to accelerate computation, already became popular in many application areas during last several years. Digital signal processing (DSP) is a significant application domain in RC. A number of papers addressed the DSP reconfiugrable applications. In [9], Lund et al discussed a flexible, reconfigurable convolution decodeing system using FPGAs. Kim et al. in [10] described an adaptive antenna signal processing algorithm for improved direction-of-arrival estimation. In the applications of image and video, reconfigurable computers also have been used successfully for accelerating low-level image processing algorithms such as local neighborhood functions. Moreover, cryptographic algorithms are particularly well suited to reconfigurable logic implementations [9]. In [12], a fully pipelined AES encryption processor mapped to the Xilinx Virtex-II Pro achieved up to 21.5Gb/s while the highest performance software implementations have been reported at 580 Mb/s. Bioinformatics also can benefit a lot from RC technology. Various generic algorithms have been mapped to hardware. For instance, the FPGA implementation of a software algorithm was

proposed by Yamaguchi, Maruyama and Konagaya [11]. Dynamic Programming (DP) algorithms were implemented on the Splash and Splash 2 FPGA system implementations in the beginning of the 90's[9]. RC has demonstrated great promise in diverse areas.

1.3 Reconfigurable Supercomputing

Due to the success of RC applications, supercomputer corporations are applying RC technology into supercomputing applications. Cray and Silicon Graphics have launched products including FPGAs used as accelerators. Existing supercomputing applications need to be modified to take advantage of RC. Urban road traffic simulation [13] is one representative example of reconfigurable supercomputing applications. The authors partitioned the simulation application between hardware and software. This simulation is a simplified version of the TRANSIMS [14] road network simulator. Since the TRANSIMS uses cellular automaton (CA) model, a high parallel computational model [13], the simulation has the potential to use the FPGA in some parallel executions. By providing hardware/software co-design in supercomputing application, the speedup over software reaches 34X including the communication cost. Therefore, reconfigurable supercomputing promises significant performance improvement, which has been attracting researchers in various computation fields.

1.4 Motivation

Performance acceleration is always one of the significant topics in scientific and engineering applications. This thesis is one of the numerous tires for the speedup purpose. We are trying to construct valuable hardware implementations of pseudorandom number generators suitable for parallel computing simulations. A well known random number generator software library – Scalable Parallel Random Number Generator Library (SPRNG), which is widely used in Monte Carlo Simulation, is chosen for hardware implementation. Since no one has implemented the SPRNG software library in hardware

before, SPRNG's Parallel Additive Lagged-Fibonacci Generator (PALFG) [1] is firstly implemented among all the random number generators provided by SPRNG library because of its good statistical properties and popularity.

2.Background Review

The common random number generators are reviewed in this chapter. The sections are organized as follows. Firstly current random number generators that have been implemented in FPGA are presented. Then the Scalable Parallel random number generator (SPRNG) library is introduced including all the random number generators it provided. Finally the Diehard test results of modified Additive Lagged Fibonacci random number generators in SPRNG are detailed listed so that statistics of several common random number generators are able to be compared.

2.1 FPGA-based Random Number Generators (RNG)

Random number generators are widely applied in diverse FPGA applications such as key generations in cryptography and genetic algorithms. In order to meet special requirements of RNG in applications such as high quality, fast generation rate, long period or either two of them, different RNGs have been chosen and developed. The most common FPGA-based RNGs include Linear Feedback Shift Register (LFSR), Multiple LFSR, Cellular Automata (CA), Multiple CA, Mersenne Twister and True RNG.

2.1.1 LFSR RNG

Linear feedback shift registers RNG is widely used in FPGA applications, as its structure is simple and efficient in hardware design. A Linear feedback shift register is a shift register whose input bit is a linear function of its previous state [16]. The linear function is usually exclusive-or and inverse-exclusive-or. For an LFSR RNG of length n, the function can be expressed as Equation 2.1.

$$f(\vec{x}) = c_0x_0 + c_1x_1 + c_2x_2 + \ldots + c_{n-1}x_{n-1} = \sum_{i=0}^{n-1} c_i x_i \quad \text{Equation - 2.1}$$

The obvious weakness of this type of RNG is that the statistics of LFSR RNG are not good since there is 50% probability that the value at next time slot can be predicted. Besides the LFSR period, $2^n - 1$, is not long, which is not a feature of good random number generators.

2.1.2 Multiple LFSRs

One method of obtaining better statistical results of the LFSR of length n is to allow the LFSR to run for n cycles before reading another number [17]. However it limits the generation rate and it is not explored any further. An equivalent result can be obtained by implementing n LFSRs of length m and using a single bit from each LFSR at each LFSR at each time step. The multiple LFSRs require the seed of each LFSR to be independent. As far as the FPGA implementation is concerned, in Xilinx Virtex FPGA, implementing a long shift register is not sufficient because the look up tables can implement a 16 bit shift register very easily, but longer shift registers require more extensive routing resources[17].

2.1.3 Cellular Automata RNG

Cellular Automata RNG is another popular RNG for hardware implementation. The CA algorithm is defined as follows. A one dimension Cellular Automata contains a string of cells. Each cell has two neighbors – left and right. For the first cell, its left neighbor is the last cell in the string as shown in Figure 1. Similar, the last cell's right neighbor is the first cell. At each time step, the value of any cell c is given by a rule. For example, the rule can be defined as $c_{t+1} = c_t \oplus left_t \oplus right_t$ where $\oplus$ denotes the exclusive OR function. CA RNGs presents good three main statistical features no period, independence and high dimensionality [18]. However, its speed is not satisfying and can't compete with LFSR, linear congruencial generators (LCG) and other generators. As the same case of multiple LFSR RNG, statistics of the results should be much better.

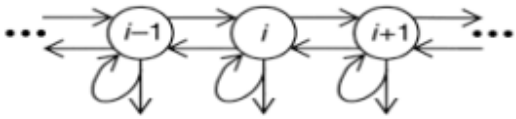

Figure 1 Cellular Automata Random Number Generator

2.1.4 Mersenne Twister RNG (MT)

Mersenne Twister (MT) RNG is developed by Makoto Matsumoto and Takuji Mishimura in 1996/1997[19]. It passes numerous tests for statistical randomness, including the stringent Diehard Tests and it is widely applied for Monte-Carlo simulations. The Mersenne Twister is a form of linear feedback shift register with an extremely long period of $2^{19937}-1$ and 623 dimensional equidistribution. Its equation is expressed as follow:

$$X_{k+n} := X_{k+m} \oplus (X_k^u \mid X_{k+1}^l)A$$

where the $\oplus$ symbol denotes the exclusive or operation (XOR), n= 624, m=397, and X^u, X^l being X with upper and lower bit masks applied.

Due to MT simple, long period and excellent statistical properties, it has been implemented on FPGAs and has been successfully applied in various applications [22] [23].

2.1.5 True RNG

All the random number generators considered so far are pseudorandom number generators. Pseudorandom number generators use deterministic processes to generate a series of outputs from an initial seed state. Unlike pseudo random number generator, a true random number generator (TRNG) uses a non-deterministic source to produce randomness. Most operate by measuring unpredictable natural processes such as thermal noises, atmospheric noise and electronic noise. The TRNG produces reasonable and satisfactory random numbers. However, surprisingly, few FPGA hardware implementations of TRNGs have been reported. [19] was reported FPGA implementation of a TRNG by Fischer and Drutarovsky using a variation of oscillator sampling. In [19], the TRNG, which employs

oscillator phase noise, was implemented. However, although TRNG provides excellent statistics that pseudorandom number generator can't compete with, its sampling a source of noise are often slow [20] so that they are not quite applicable to high speed system.

2.2 Scalable Parallel Random Number Generator (SPRNG) Library

SPRNG software was funded from DARPA and was especially designed for large-scale, parallel Monte Carlo Simulations. Until now SPRNG library has been widely applied into this field as its compact codes, large scalability, and great statistical properties [25] [26].

To be parallel and scalable, SPRNG mainly focuses on the methods based on parameterization instead of splitting methods such as the leap-frog or blocking method. In the sequence splitting method, a serial random number stream is partitioned into non-overlapping contiguous subsequences. Parameterization method identifies a parameter in the underlying recursion of a serial random number generator that can be varied. Each valid value of this parameter leads to recursion that produces a unique full-period stream of random numbers[24].

SPRNG provides parameterized versions of the pseudorandom number generators: combined Multiple Recursive Generator, 48-bit and 64-bit linear congruential generators, modified additive and multiplicative lagged-Fibonacci generators, and Prime Modulus Linear Congruential Generator. In the next parts, these generators properties including recursion functions, period and independence streams will be briefly introduced.

2.2.1 Combined Multiple Rescursive Generator

This generator follows the relation as Equation 2-4:

$$Z(n) = X(n) + Y(n) \times 2^{32} \mod 2^{64} \qquad \text{Equation- 2.4}$$

where $X(n)$ is the sequence generated by the 64-bit linear congruential generator and $Y(n)$ is prime modulus multiple recursive generator which is expressed in Equation 2-5

$$Y(n) = 107374183 \times Y(n-1) + 104480 \times Y(n-5) \pmod{2147483647} \quad \text{Equation 2-5}$$

The period of this RNG is 2^{219} and the number of the independent streams available reaches to over 10^8.

2.2.2 48 bit and 64 bit Linear Congruential Generator with Prime Addend

This RNG recurrence equation is expressed in Equation 2-6:

$$X(n) = aX(n-1) + p \mod M \quad \text{Equation 2-6}$$

where $X(n)$ is the nth term in the sequence, p is a prime number and a is the multiplier. When $M = 2^{48}$, the RNG is 48 bit LCG with Prime Addend. When $M = 2^{64}$, the RNG is 64 bit LCG.

The period of 48 bit LCG is 2^{48} and its number of distinct streams is 2^{19} while 64 bit LCG is 2^{64} and its number of distinct streams available is over 10^8.

2.2.3 Prime Modulus Linear Congruential Generator

This generator's relation is expressed as Equation 2-7 . The maximal period is $2^{61} - 2$ and the number of distinct streams available is roughly 2^{58}. For each stream, the a differs.

$$X(n) = a \times X(n-1) \mod (2^{61} - 1) \quad \text{Equation 2-7}$$

2.2.4 Multiplicative Lagged Fibonacci Generator

This generator is following the equation 2-8

$$X(n) = X(n-k) \times X(n-l) \mod M \quad \text{Equation 2-8}$$

where l and k are called the lags of the generator and it is defined that l is larger than k. In SPRNG, M is chosen to be 2^{64}. The period of this generator is $2^{61}(2^{l}-1)$ and the number of distinct streams available is $2^{[63(l-1)-1]}$.

The random number sequence obtained is determined by the l initial values of the sequence X. It is important to use the same original seed to initialize l initial values for each stream so that all the streams with this generator are independent.

2.2.5 Modified Additive Lagged Fibonacci Generator (PALFG)

Normally, Additive Lagged Fibonacci Generator (PALFG) is defined as Equation 2-9

$$X(n) = X(n-k) + X(n-l) \mod 2^{32} \qquad \text{Equation 2-9}$$

where l and k are called the lags of the generator and it is defined that l is larger than k. In the SPRNG, the authors modified the Lagged Fibnocci Random Number Generator to improve the statistical property. The modified ALFG (PALFG) has the following relations:

$$Z(n) = X(n) \oplus Y(n) \qquad \text{Equation 2- 10}$$

Where $X(n) = X(n-k) + X(n-l) \mod 2^{32}$; $Y(n) = Y(n-k) + Y(n-l) \mod 2^{32}$;

SPRNG provides total 11 types of random number generator as shown in Table 1. PALFG has long period, which is $2^{31}(2^{l}-1)$ and its number of distinct streams available are $2^{[31(l-1)-1]}$. When l is 1279, the period is approximately 2^{1310} and impressively, this random number generator gives 2^{39617} distinct streams. This shows that this random number generator has great scalability and parallelism that are attractive to those parallel huge simulations.

Table 1. Parameterized PALFG Generators Provided By SPRNG

l	*k*	*m*
17	5	32
31	6	32
55	24	32
63	31	32
127	97	32
521	168	32
521	253	32
607	273	32
607	334	32
1279	418	32
1279	861	32

2.3 Statistic Analysis

2.3.1 Diehard Test

Diehard tests are statistical tests for measuring the quality of a set of random numbers [27]. They are widely accepted as one of most authoritative tests. Most random number generators demonstrate its statistics properties using Diehard test results. Diehard tests have 12 common tests, which are introduced briefly in the following.

1. Birthday Spacing: The spacing between the random points should be asymptotically poisson distributed.
2. Overlapping Permutations: Based on the analysis results of sequences of five consecutive random numbers, the 120 possible orderings should occur with statistically equal probability.
3. Ranks of matrices: Part bits from some random numbers are chosen to form a matrix {0,1}. The rank of the matrix will be counted
4. Monkey Tests: Some bits in the sequences are combined as “words”. The number of the overlapping words in a stream should follow a known distribution.
5. Count the 1s: the ‘1’ bits in each chosen byte are counted and the count number will be converted to “letter” and, five letters are combined together into a word, the occurrence of five-letter word will be counted.
6. Parking Lot Test: In a square of side 100, randomly “park” a car using increasing unit circles. If the circle overlaps an existing one, try again. After 12000 tries, the number of successfully “parked” circles should follow a certain normal distribution.
7. Minimum Distance Test: In a square of side 10,000, find the minimum distance between the pairs. The square of this distance should be exponentially distributed with a certain mean.

8. Random Spheres Test: Choose 4,000 points in a cube of edge randomly and center a sphere on each point. The smallest sphere's volume should be exponentially distributed with a certain mean.
9. The Squeeze Test: Multiply 2^{31} by random floats on [0,1) until reaching 1 and repeat the above procedure 100,000 times. The number of floats needed to reach 1 should follow a certain distribution.
10. Overlapping Sums Test: Every 100 consecutive floats are added and the sums should be normally distributed with characteristic mean and sigma.
11. Run Test: Count the ascending and descending runs in a long sequence. The count number should follow a certain distribution.
12. The Craps Test: Play 200,000 games of craps and find the number of wins and the number of throws per game. Each count should follow a certain distribution.

2.3.2 Grading Rules

To simplify the diehard results and make the results readable and understandable for everyone, Johnson [29] introduced a grading rule as follows. He assigned a score to a *p*-value as good, bad or suspect, three different grade. If $p > 0.998$ then it is classified as bad. If $0.95 < p < 0.998$ then it is classified as suspect. All other *p*-values are classified as good. Every bad *p*-value scores 4, every suspect *p*-value scores 2 and good *p*-values score zero. For each RNG, the scores for each test were summed, and the total for each RNG is the sum of all the test scores for that RNG. Using this scheme, high scores indicate a poor RNG and low scores indicate a good RNG. In this chapter, this grading rule is applied to evaluate the statistics of the random number generators.

2.3.3 Statistics of Common Hardware RNG

Peter Martin [17] presented the Diehard test results of several common implemented hardware random number generators as mentioned in the above, which is shown in Table 2. The max score means that the random number generator doesn't pass the test. From the table we can see that true random number generator has the best statistical property. However, LFSR, which is the most widely used in the hardware implementation, has the worst statistics compared with others. Therefore, although LFSR is simple and easy to be implemented in the hardware, in most hardware applications that highly demand good random number generators such as Monte Carlo simulations, LFSR is not an ideal choice.

2.3.4 Statistics of PALFG in SPRNG

In this section, I presented PALFG diehard tests results followed by the above grading rule. The test samples are obtained from SPRNG library and the number of test samples reaches about 3 million 32-bit random numbers.Table 3 and Table 4 listed all the 11 parameter sets of PALFG statistical results. From the experiment results, each PALFG type has almost the same statistical property, whose grade is around 240. Except True RNG diehard results, the PALFG ranks 2nd in all the pseudorandom number generators listed above, while 32 LFSR ranks highest among them. Although 32 LFSR provides better property in the Diehard tests, the 32 LFSR is not designed for parallel and scalable parallel simulations. [16] mentioned that SPRNG provided a test suite that implemented all the tests described by Knuth[30] and except 48-bit LCG all other SPRNG random number generators passed the Diehard tests. In conclusion, from the Diehard test results, PALFG is a good random number generator and it is worth being implemented in hardware as far as the performance speedup of scalable and parallel simulations is concerned. Because it is widely used, an accelerated SPRNG will help a diversity of computational science applications.

Table 2. Diehard Test Results for Common RNG in Hardware [17]

Test	Max Score	LFSR	EQG	32LFSR	IDCA	32CA	True
Birthday	36	36	8	2	0	8	0
Overlapping Permutation	8	8	0	4	8	8	0
Binary Rank 32 * 32	8	8	2	8	2	6	0
Binary Rank 6x	104	104	40	8	140	70	4
Bitstream	80	80	0	0	80	80	4
Overlapping Pairs tests	328	328	188	94	328	320	6
Count the ones(stream)	8	8	8	8	8	8	0
Count the Ones (specific)	100	100	42	30	100	100	2
Parking Lot	44	4	0	0	4	2	0
Minimum Distance	4	4	0	4	4	4	0
3D spheres	84	4	0	2	4	2	4
Squeeze	4	4	0	0	4	4	0
Overlapping Sums	44	44	0	0	6	0	2
Runs	16	16	0	2	16	8	0
Craps	8	8	0	0	8	12	0
Total	876	756	288	162	676	640	22

Table 3. Diehard Test Results for PALFG I

Test	Max Score	LFG (1279,861)	LFG (17, 5)	LFG (31,6)	LFG (55,24)	LFG (63, 31)
Birthday	36	4	4	4	4	4
Overlapping Permutation	8	0	0	2	0	0
Binary Rank 32 * 32	8	8	8	8	8	8
Binary Rank 6x	104	12	6	8	10	8
Bitstream	80	80	80	80	80	80
Overlapping Pairs tests	328	12	24	22	16	20
Count the ones(stream)	8	8	8	8	8	8
Count the Ones (specific)	100	6	8	6	6	10
Parking Lot	44	44	44	44	44	44
Minimum Distance	4	4	4	4	4	4
3D spheres	84	4	4	4	4	4
Squeeze	4	--	--	--	--	--
Overlapping Sums	44	44	44	44	44	44
Runs	16	0	0	2	0	0
Craps	8	8	8	8	8	8
Total	876	234	242	244	236	244

Table 4. Diehard Test Results for PALFG II

Test	Max Score	LFG (127, 97)	LFG (521, 353)	LFG (521,168)	LFG (607, 334)	LFG (607, 73)	LFG (1279,418)
Birthday	36	4	4	8	4	4	4
Overlapping Permutation	8	0	4	0	0	2	0
Binary Rank 32 * 32	8	8	8	8	8	8	8
Binary Rank 6x	104	6	4	4	6	6	4
Bitstream	80	80	80	80	80	80	80
Overlapping Pairs tests	328	18	20	20	20	22	22
Count the ones(stream)	8	8	8	8	8	8	8
Count the Ones (specific)	100	8	4	4	4	8	6
Parking Lot	44	44	44	44	44	44	44
Minimum Distance	4	4	4	4	4	4	4
3D spheres	84	4	4	4	4	4	4
Squeeze	4	--	--	--	--	--	--
Overlapping Sums	44	44	44	44	44	44	44
Runs	16	0	0	0	4	0	0
Craps	8	8	8	8	8	8	8
Total	876	238	238	238	240	240	234

3. SPRNG Design

The design of SPRNG's Parallel Additive Lagged-Fibonacci Generator (PALFG)[1] is described in this chapter. The sections are organized as follows. First, three design criteria are presented. Second, the PALFG execution steps in software are elaborated in the order that they are conducted. Next, the hazards in design are proposed. Finally, a neat and efficient design flow, as well as the solutions to the hazards, is presented in detail.

3.1 Design Criteria

As the random number generation hardware implementation is targeted to applications in high performance scientific computing, three important design criteria are considered. First, the hardware implementation should produce random numbers identical to those of the corresponding software SPRNG generator for all values of the parameters *l* and *k* supported by SPRNG, because the implementation will be accepted for scientific simulations only if an exact reproduction of the SPRNG generator is provided. Thus, the final implementation should be rigorously tested to verify the output results.

Second, the design should be general. It should be suitable for all the parameters *l* and *k* supported by SPRNG and also flexible to accommodate future expansion to representations larger than 32 bits. Moreover, to achieve high performance, we prefer an implementation that produces one random number per clock cycle.

Third, the design should be sufficiently portable and as simple as possible. A compact design allows the remaining FPGA logic resources to be utilized for other purposes.

3.2 PALFG Execution in SPRNG

It is observed that there is certain correlation in the Additive Lagged-Fibonacci number generator (ALFG). SPRNG modified the ALFG, called PALFG, to avoid this correlation by the equation:

$$Z = X \oplus Y$$

where X and Y are two distinct ALFG sequences. Prior to the XOR operation, x_n in X is modified by setting the least significant bit to zero and by right shifting y_n in Y to the right by one bit. Following the XOR operation, the least significant bit is ignored to yield a 31-bit uniform random integer. The following pseudo code describes the PALFG algorithm implementation in SPRNG.

```
1. Initialization Seeds

2. Build up two arrays x and y with length L to store L 32-bit random numbers. Each of
   arrays has two pointers (hptr and lprt)

3. Generate an unsigned integer random number

   lptr = hptr+k; // calculate low pointer address

   if (lptr >= l) lptr - = l; // calculate low pointer address

   x[hptr] = x[hptr] + x[lptr]; // update X value

   y[hptr] = y[hprt] + y[lptr]; // update Y value

   if (--hptr <0) hptr = l-1; // calculate high pointer address

    z = ( (x[hptr] & $FFFFFFFE) xor ( y[hptr] >>1) ) ;    // calculate    Z

   new value = z >> 1    /*random number */

   if(--lptr    < 0) lptr = l-1; // calculate low pointer address

   x[hptr] = x[hptr] + x[lptr]; // update X value

   y[hptr] = y[hprt] + y[lptr]; // update Y value
```

A {17,5} random number generator example in SPRNG is described here to illustrate this pseudo code and propose an efficient design. As X and Y sequences are produced in the same way, Table 5 only lists X sequence execution steps. From the table, it is observed that two add operations execute simultaneously in each iteration and the sum result of stage 1 is selected as the one of the two operands in the next XOR step.

3.3 PALFG Design Hazards

Table 5 infers that there are four execution steps -- Read, Accumulation, Write, and Selection, that should be completed in one clock cycle to produce one random number per clock cycle. Hence, two types of hazards can occur: structural hazards and data hazards.

Structural Hazard: Four data values have to be read and two data values have to be written in the same clock. The requirement for a six-port memory results in a potential structural hazard because the FPGAs we are using don't include 6-port RAMs.

Data Hazard: The design must work correctly with respect to potential data hazards. In table 5, since X_0 has to be available in the first and third iteration, the new calculated X_0 must be written back to memory in the second iteration. Therefore, for the {17,5} generator, its design at most has two pipeline steps. Read and accumulation operations are executed in the same step, which greatly influences the performance of the design. Three pipeline stages (read, accumulate, write back) in this case are preferred. Unfortunately, a Read After Write (RAW) data hazard will occur. Hence, the pipeline requirements coincide with the lag *k*. For the general design to be applicable for all SPRNG parameter sets, the generators that have the smallest lag *k*, {17,5} and {31, 6}, have to be satisfied with a number of pipeline steps to avoid the RAW hazard.

Thus, these two hazards have to be investigated and be avoided in the design to achieve the optimum performance.

Table 5. {17,5} PALFG Execution Sequence

Iteration	Stage1	Stage2	Number Selected
Initialize	--	--	
0	$X_0 = X_0 + X_{12}$	$X_1 = X_1 + X_{13}$	X_0
1	$X_2 = X_2 + X_{14}$	$X_3 = X_3 + X_{15}$	X_2
2	$X_4 = X_4 + X_{16}$	$X_5 = X_5 + X_0$	X_4
3	$X_6 = X_6 + X_1$	$X_7 = X_7 + X_2$	X_6
4	$X_8 = X_8 + X_3$	$X_9 = X_9 + X_4$	X_8
5	$X_{10} = X_{10} + X_5$	$X_{11} = X_{11} + X_6$	X_{10}
6	$X_{12} = X_{12} + X_7$	$X_{13} = X_{13} + X_8$	X_{12}
7	$X_{14} = X_{14} + X_9$	$X_{15} = X_{15} + X_{10}$	X_{14}
8	$X_{16} = X_{16} + X_{11}$	$X_0 = X_0 + X_{12}$	X_{16}
9	$X_1 = X_1 + X_{13}$	$X_2 = X_2 + X_{14}$	X_1
10	$X_3 = X_3 + X_{15}$	$X_4 = X_4 + X_{16}$	X_3
11	$X_5 = X_5 + X_0$	$X_6 = X_6 + X_1$	X_5

Table 5 Continued

Iteration	Stage1	Stage2	Number Selected
12	$X_7 = X_7 + X_2$	$X_8 = X_8 + X_3$	X_7
13	$X_9 = X_9 + X_4$	$X_{10} = X_{10} + X_5$	X_9
14	$X_{11} = X_{11} + X_6$	$X_{12} = X_{12} + X_7$	X_{11}
15	$X_{13} = X_{13} + X_8$	$X_{14} = X_{14} + X_9$	X_{13}
16	$X_{15} = X_{15} + X_{10}$	$X_{16} = X_{16} + X_{11}$	X_{15}
17	$X_0 = X_0 + X_{12}$	$X_1 = X_1 + X_{13}$	X_0

3.4 PALFG Execution Sequence

According to the modified algorithm of ALFG in SPRNG, Figure 2 shows PALFG execution sequence. First, two independent 32-bit ALFG units (labeled as X and Y) of identical architecture coupled by a 32-bit exclusive-or unit generate two independent pseudorandom numbers. Prior to XOR operation, the output of X is modified by setting the least significant bit to zero and by right shifting the output of Y by one bit. The overall PALFG sequences are quite simple while the ALFG unit is more complex since care should be taken in this module to avoid the hazards and implement an efficient general random number generator for all parameters in SPRNG.

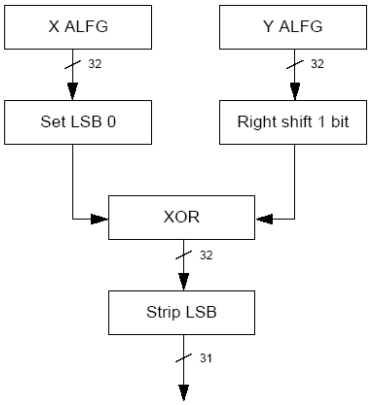

Figure 2. PALFG Execution Sequence

3.5 ALFG Unit Design

To avoid the structural hazard, we can easily build up a virtual six-port memory by accommodating multiple dual port memory resources. Compared with the structural hazard, the solution to the data hazard is more complex if we want to satisfy the three design criteria. For instance, because the pipeline requirements depend on the lag *k*, the short pipeline of the {17,5} generators poses serious timing issues. This is because the SPRNG PALFG implementation must advance each ALFG forward by two recurrence steps every time that a random number is generated. So if we can design the hardware such that the number of pipeline steps is independent of the *k*, the data hazard will be avoided. The approach contained in this thesis is that rather than try to have dependent data updated and rewritten to memory so that it is available for reading $\left\lfloor \frac{k}{2} \right\rfloor$ clocks later, it makes sense to go ahead and satisfy the dependency first before writing the data back to memory, which is called a look ahead approach or forwarding. In other words, we can make values x_i as operands appear in the two stages of the same iteration by computing recurrence steps out of order as long as the data dependencies are satisfied. To fully illustrate this design, the {17, 5} generator is considered once again. For the {17, 5}

generator, we can see from the table 5 that in the iteration zero, the output X_0 of the first stage resulting from the operation $X_0 = X_0 + X_{12}$ is used in the second stage of the second operation $X_5 = X_5 + X_0$. By advancing two operations of the second stages, the X_0 value appears in both stages of the same iteration. Then the updated X_0 doesn't need to be written back to the memory until after $\left\lfloor \frac{l}{2} \right\rfloor$ clocks, which is much longer than $\left\lfloor \frac{k}{2} \right\rfloor$. Then the number of pipeline steps can be extended and the data hazard is avoided. Table 6 lists the example execution flow steps of {17, 5} generator. Given the above idea, our ALFG architecture is proposed. In the design, three separate memory resources (denoted XA, XB and XC) are used to support the three required simultaneous reads from memory. The corresponding schematic design is presented in Figure 3. A1 denotes the output of the first stage (recurrence) operation, which is just the updated value of X_n. A1 serves as the input for the second stage. A2 is the output of the second operation. At the end of the iteration, the outputs A1 and A2 are written to memories. The output A1 is written to XC because it will need to be read from XC after $\left\lfloor \frac{l-k-1}{2} \right\rfloor$ clocks. Similarly, the output A2 is written simultaneously to both XA and XB and is not used again until after $\left\lfloor \frac{l-k-(k-1)}{2} \right\rfloor$ if $(l > 2k)$ or $\left\lfloor \frac{2l-k-(k-1)}{2} \right\rfloor$ if $(l < 2k)$. In this way, we avoid having to perform more than one write to each memory resource per clock. The desired 32-bit random numbers are obtained by tapping the A1 output, which corresponds to the column labeled in "Stage 1" in Table 6. This cycle will continue indefinitely and will generate all values in the random number sequence, though slightly reordered. It is also noted that for this design to work, several values must be pre-initialized before execution begins. We will describe the initialized algorithms in the following sections.

Table 6. Look Ahead Approach on {17,5} PALFG

Iteration	Stage1	Stage2	Number Selected
Initialize	--	$X_1 = X_1 + X_{13}$	
Initialize	--	$X_3 = X_3 + X_{15}$	
0	$X_0 = X_0 + X_{12}$	$X_5 = X_5 + X_0$	X_0
1	$X_2 = X_2 + X_{14}$	$X_7 = X_7 + X_2$	X_2
2	$X_4 = X_4 + X_{16}$	$X_9 = X_9 + X_4$	X_4
3	$X_6 = X_6 + X_1$	$X_{11} = X_{11} + X_6$	X_6
4	$X_8 = X_8 + X_3$	$X_{13} = X_{13} + X_8$	X_8
5	$X_{10} = X_{10} + X_5$	$X_{15} = X_{15} + X_{10}$	X_{10}
6	$X_{12} = X_{12} + X_7$	$X_0 = X_0 + X_{12}$	X_{12}
7	$X_{14} = X_{14} + X_9$	$X_2 = X_2 + X_{14}$	X_{14}
8	$X_{16} = X_{16} + X_{11}$	$X_4 = X_4 + X_{16}$	X_{16}
9	$X_1 = X_1 + X_{13}$	$X_6 = X_6 + X_1$	X_1

Table 6 Continued

Iteration	Stage1	Stage2	Number Selected
10	$X_3 = X_3 + X_{15}$	$X_8 = X_8 + X_3$	X_3
11	$X_5 = X_5 + X_0$	$X_{10} = X_{10} + X_5$	X_5
12	$X_7 = X_7 + X_2$	$X_{12} = X_{12} + X_7$	X_7
13	$X_9 = X_9 + X_4$	$X_{14} = X_{14} + X_9$	X_9
14	$X_{11} = X_{11} + X_6$	$X_{16} = X_{16} + X_{11}$	X_{11}
15	$X_{13} = X_{13} + X_8$	$X_1 = X_1 + X_{13}$	X_{13}
16	$X_{15} = X_{15} + X_{10}$	$X_3 = X_3 + X_{15}$	X_{15}

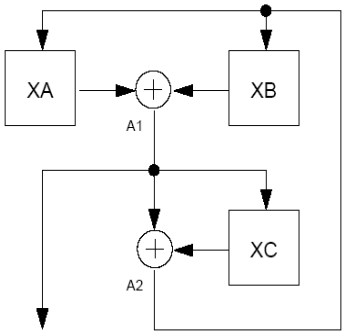

Figure 3. ALFG Schematic Design, *k odd*

There is one detail that has not been described yet. The ALFG design presented in Figure 3 is only valid for sets of parameters *l* and *k* where *k* is odd. The oddness of k conveniently arranges for data dependencies to always occur in the second stage. For generators such as the {31,6}, where k is even, the situation is reversed and the dependencies all occur in the first stage. This is evident in the list of memory reads for the {31, 6}, Table 7. In iteration 0, we update the value X_0, but X_0 is needed again in iteration 3, stage 1. This means that we cannot cascade two steps as before to remove the dependency problem. However, we can arrange stage 2 to take advantage of the value read from XB during stage 1. Thus, iteration 0, the value X_{25} is used to update X_0 in stage 1 while X_{25} itself is simultaneously updated in stage 2. The updated value X_{25} is then available the next time it is needed (iteration 28). The ALFG schematic design for even *k* is presented in Figure 4. To ensure that updated values are available from the appropriate resources when needed, the ouput A1 is written back to XB. A2 is simultaneously written back to both XA and XC. Thus, we can maintain a pattern of data flow similar to that of the even *k* design.

Table 7. Look Ahead Approach on {31,6} PALFG

Iteration	Stage1	Stage2	Number Selected
Initialize	--	$X_1 = X_1 + X_{26}$	
Initialize	--	$X_3 = X_3 + X_{28}$	
Initialize	--	$X_5 = X_5 + X_{30}$	
Initialize	--	$X_7 = X_7 + X_1$	
Initialize	--	$X_9 = X_9 + X_3$	
Initialize	--	$X_{11} = X_{11} + X_5$	
Initialize	--	$X_{13} = X_{13} + X_7$	
Initialize	--	$X_{15} = X_{15} + X_9$	
Initialize	--	$X_{17} = X_{17} + X_{11}$	
Initialize	--	$X_{19} = X_{19} + X_{13}$	
Initialize	--	$X_{21} = X_{21} + X_{15}$	
Initialize	--	$X_{23} = X_{23} + X_{17}$	

Table 7 Continued

Iteration	Stage1	Stage2	Number Selected
1	$X_0 = X_0 + X_{25}$	$X_{25} = X_{25} + X_{19}$	X_0
2	$X_2 = X_2 + X_{27}$	$X_{27} = X_{27} + X_{21}$	X_2
3	$X_4 = X_4 + X_{29}$	$X_{29} = X_{29} + X_{23}$	X_4
4	$X_6 = X_6 + X_0$	$X_0 = X_0 + X_{25}$	X_6
	.	.	
	.	.	
26	$X_{21} = X_{21} + X_{15}$	$X_{22} = X_{22} + X_{17}$	X_{21}
27	$X_{23} = X_{23} + X_{17}$	$X_{24} = X_{24} + X_{19}$	X_{23}
28	$X_{25} = X_{25} + X_{19}$	$X_{26} = X_{26} + X_{21}$	X_{25}
29	$X_{27} = X_{27} + X_{21}$	$X_{28} = X_{28} + X_{23}$	X_{27}
30	$X_{29} = X_{29} + X_{23}$	$X_{30} = X_{30} + X_{25}$	X_{29}

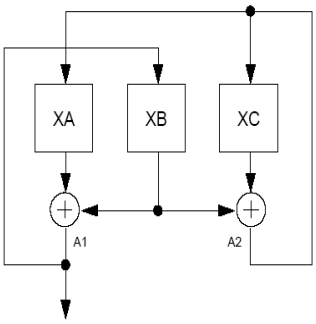

Figure 4.ALFG Schematic design, *k* even

A completely general PALFG implementation would necessarily employ circuits for both cases, *k* even and odd, to be able to handle all SPRNG parameter sets.

3.6 Seed Initialization Algorithm for the ALFG design

First, let's assume that the original *l* seeds obtained from SPRNG Library algorithms are given. Also assume that the *l* seeds are numbered from 0 to $l-1$ in order. Given two design schemes for even *k* and odd *k*, the seed initialization algorithms are only slightly different. To simplify the description, the seeds numbered for odd k from 0 to *n* need to be calculated in the seed initialization algorithm using formula $X_i = X_i + X_m$, where $m = (i + l - k) \bmod l$ and $0 \le i \le n$

For even *k* ALFG unit, $n = l - k - 2$;

For odd *k* ALFG unit, $n = k - 2$;

Using the seed initialization algorithm, seeds will then be calculated to initialize the PALFG cores.

3.7 PALFG Interface Design

The PALFG design supports all the parameter sets from SPRNG. But the specific random number generator is selected based on the parameters. An interface for the PALFG module is employed when the PALFG module is embedded into an application. Figure 5 shows the general interface design for the PALFG.

In Figure 5, we can see that PALFG provides a set of input signals for external devices to control it. The external device gives high-level commands to the module and supplies the necessary PALFG inputs such as *l*, *k* and initial seed array. Buffering the commands and data sets is used to resolve synchronization issues between the external device and PALFG module. The dual port RAMs and registers are addressable by the external device so that communication between PALFG and external device is via shared dual port RAMs and registers.

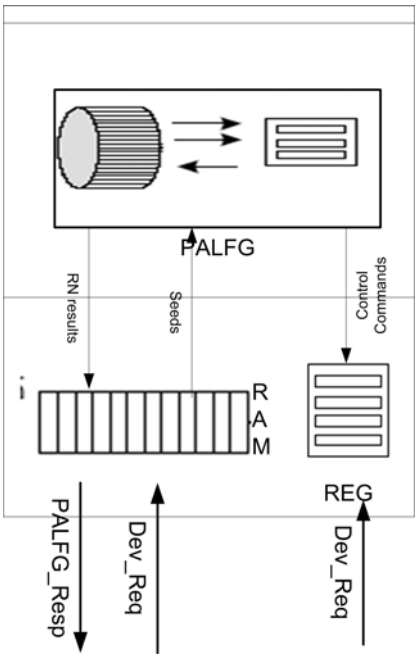

Figure 5. PALFG Interface Scheme

4. SPRNG Implementation

Following the design description, the implementation is presented in this chapter rather than the architecture specification. The implementation of three main modules comprising the PALFG including the ALFG core, execution module, and controller are detailed.

4.1 Overall PALFG Module

Both Odd-Odd PALFG and Odd-Even PALFG top-level architectures contain two ALFG cores (X and Y), one execution core and one controller. The ALFG Block implements the modified Additive Lagged Fibonacci random number generation algorithm and generates initial pseudorandom numbers. The execution core maniupulates two ALFG results to produce the final SPRNG pesudo random numbers. The controller provides a set of control and status registers that allow an external device connected to it to initialize and configure the PALFG. Figure 6 shows the architecture of the PALFG FPGA implementation. VHDL code is included in Appendix I.

4.2 Controller Block

The Controller block interfaces directly to the external devices. It provides a set of configuration and status registers to initialize and configure the PALFG FPGA.

The controller block contains a control state machine (Figure7) to handle initialization steps and parameter set configuration. This block has three states: initialization, polling the status, and output of random numbers. The block stays in an initialization state while it waits for the two ALFG blocks to finish their initialization. After initialization, the controller goes into the polling state. When the ALFG status is ‘1’, then the control block will immediately start the execution block and then goes into the output state.

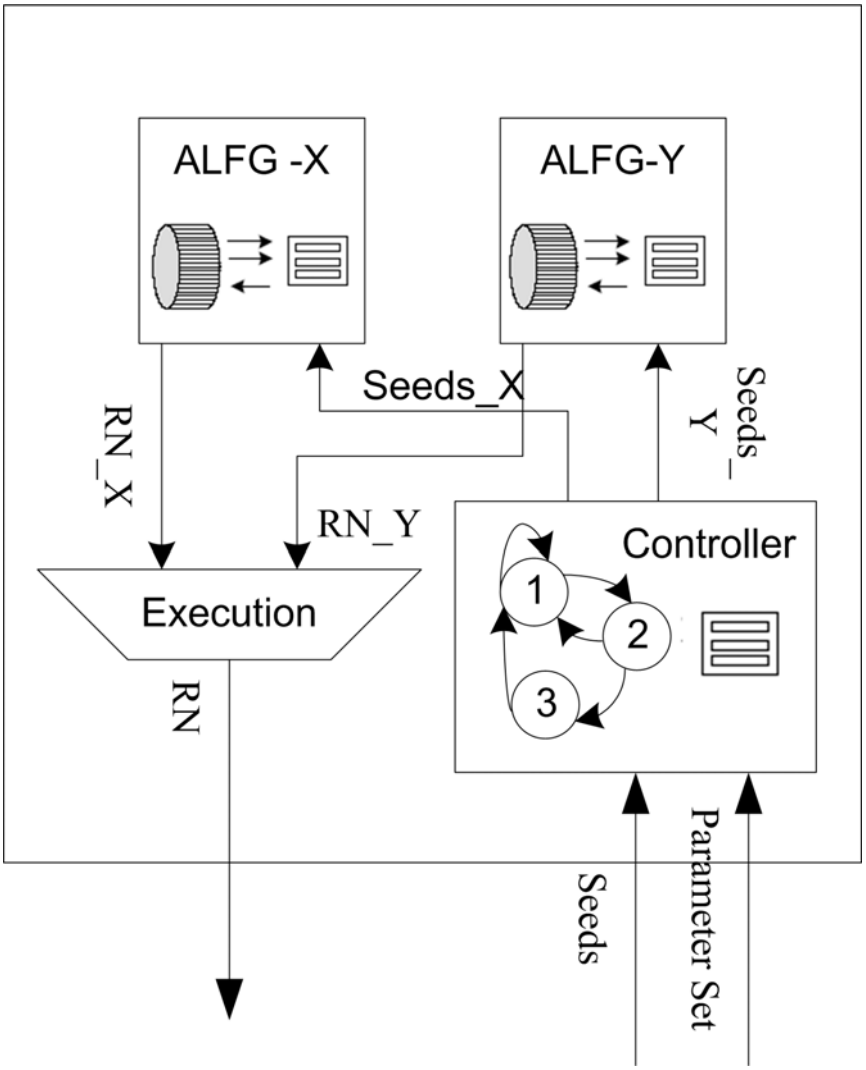

Figure 6. PALFG overall Architecture

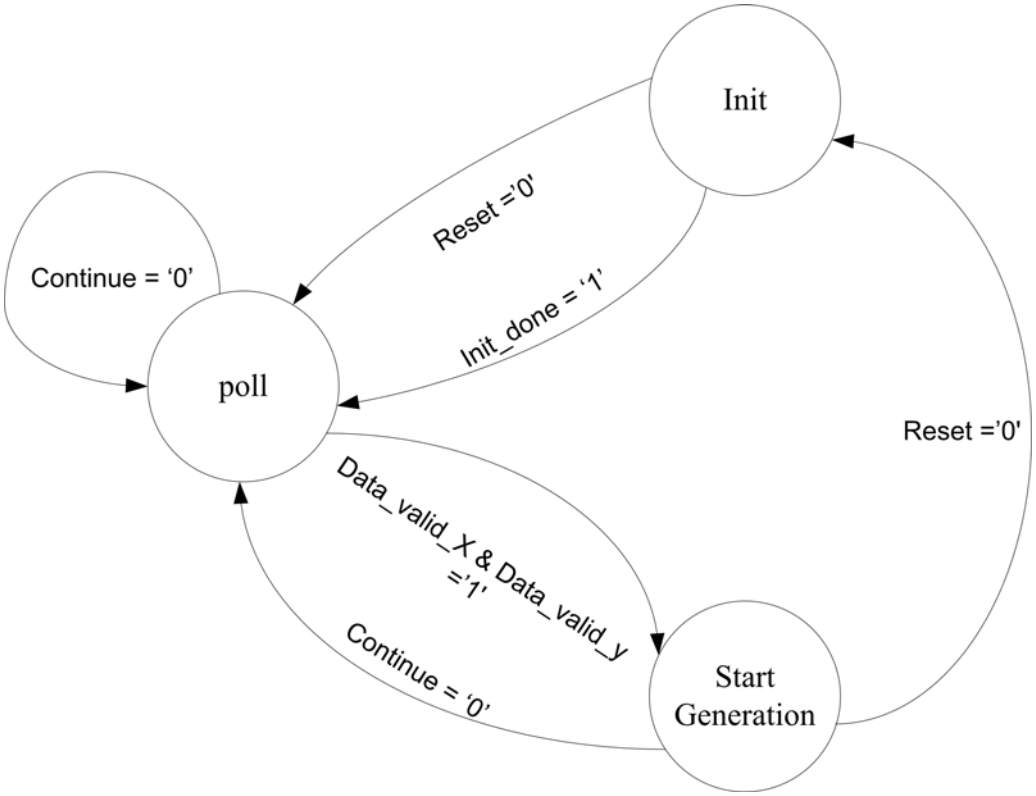

Figure 7. Controller State Machine

4.3 ALFG Block

Since the design architecture has been described in details in Chapter 3, we focus on the implementation methods in this section. In order to achieve high performance, the block takes advantage of parallelism and pipelining opportunities within the PALFG algorithm. As can be seen in the pseudo code in Chapter 3, to calculate a new array element x_n the algorithm needs to access the state RAM array values x_n, x_{n+1}, x_{n+l-k} and $x_{n+1+l-k}$. It also needs to be able to write the updated values x_n and x_{n+1}.

The algorithm is parallelized in the following ways:

- Use of dual port RAM. This allows the array to be read and written in parallel.
- Use of three copies of the RAM. This allows the state array to be read at three separate locations in parallel.

The pipelining serves one main purpose: it allows the data manipulation to be broken into smaller stages, which improves the performance. The algorithm is pipelined as four stages in odd-odd PALFG and two stages in odd-even PALFG as these two design architectures are different due to the data hazards discussed in Chapter 3. For odd-odd PALFG, the pipeline steps are RD/WR, ALU, RD/WR, ALU. For odd-even PALFG, there are only two stages: RD/WR and ALU.

The RAM array implementation is essential in this design. The RAM array is turned into a circular buffer and the circular buffer is created by implementing the read and write pointers as counters.

To accomplish the goal of general design for all parameter sets in SPRNG, the following consideration should be taken. When the parameter *l* is small, three RAM arrays (XA, XB and XC) can represent single memory resources. However, a single Block RAM resource provides only about 18 kilobits of memory on Xilinx VirtexII Pro devices. If this memory is used in a 36-bit wide by 512-bit long configuration, only parameter *l* values less than 512 may be used for the 32-bit generator. Obviously, if we are interested in 64 or 128 bit random numbers or larger values of *l* (such as 1278), we need additional RAM resources. We could extend the RAM sequentially by tacking on additional 18kb units. However, we must be careful to avoid timing issues that occur when routing data to the correct memory resources. This becomes more problematic with larger lags and greater bit widths.

So a slightly different memory-partitioning scheme was designed as shown in Figure 8. BRAM resources in 9-bit by 2048-bit configurations are used in this scheme. 32-bit values are then constructed by reading 9-bit simultaneously from each of four BRAM resources. Memory writes are accomplished in a similar manner by splitting the incoming 32-bit value into 8 bit chunks. This approach limits *l* to 2048, but this is more than sufficient for implementing all SPRNG ALFG generators. Using this approach, the bit width can now be easily tuned from 32 bits to up to widths of 512 bits or more by simply using additional BRAM units. For most generator parameters, this scheme requires slightly more BRAM but garantees that the performance will be consistent and independent of the chosen generator parameters or bit width.

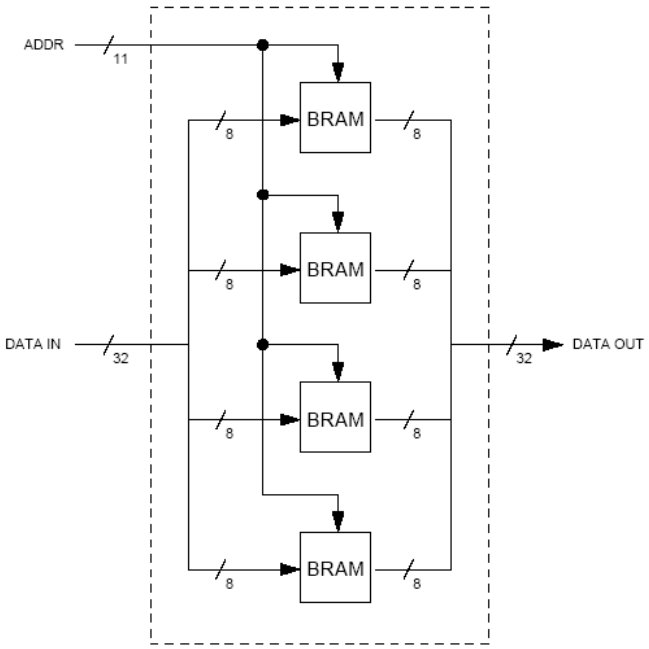

Figure 8. Memory Arrangement

5. Results

The PALFG design was designed using VHDL and each module was tested step by step. To verify the random number generated by the whole circuit identical to the SPRNG library, test benches in VHDL and software test codes for both initial seeds and results were implemented. Then the PALFG design was verified by pre-synthesis and post-layout simulation. Finally, after placing and routing, the PALFG bit stream was downloaded to the University Xilinx Program board (XUP) with Xilinx VirtexII Pro XC2VP30. The final results verified the PALFG design and are identical to SPRNG Library. In the following sections, the results, as well as verification methods, are presented.

5.1 Test Samples Generation

Although the SPRNG library provides the open source, it is not designed for the hardware verification. It doesn't provide specific functions to export seeds nor a golden random number results. So one test software application containing an Additive Lagged-Fibonacci random number generator core in SPRNG, was programmed to export the recalculated seeds and random number results to the text files. After the parameters of random number generators and the number of generators are chosen, corresponding seeds are generated using SPRNG cores and written into one text file. Figure 9 is the software interface. Source code is in Appendix II.

5.2 RTL Verification

Mentor ModelSim was used for function simulation. Test benches were applied to verify the behavior of individual modules. A Graphic User Interface (GUI) is helpful and easier for verification in this case because the results can be easily observed. The test bench is developed to get the final results (random numbers) that are shown on the wave window in ModelSim.

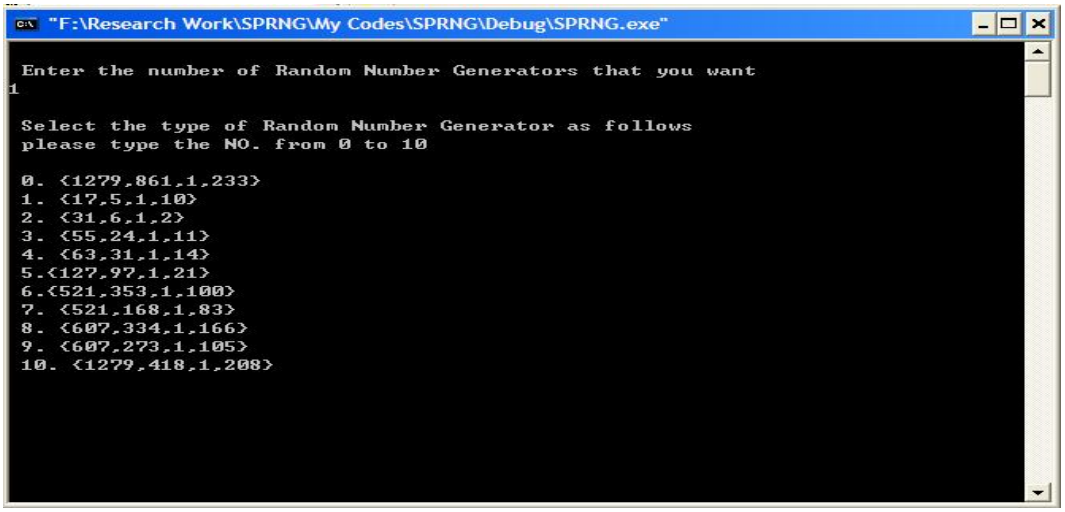

Figure 9. Software Interface

As the seeds and parameter sets of random number generators need to be set before the hardware executes, the test bench was written to simulate an external device to transfer the control signal and essential data. Considering that the circuit has to been verified for all the eleven parameter sets, it is tedious and impractical to have 11 separate test benches. Therefore a readfile module is included, which is responsible to read parameter sets and input seeds from a text file. In the test procedure, the readfile component reads the first two lines to set the parameters (l,k) and the rest of data as seeds input. After the seeds and parameters are sent to the circuit, the circuit starts to generate random numbers and the results then appear in the GUI. Figure 10 shows one of the simulation results. In the figures, the values in wave windows highlighted in red circles are the random numbers produced by the PALFG module. Through comparison with the golden results in Notepad window, the circuit has been logically verified. All the simulation results are identical to the golden ones.

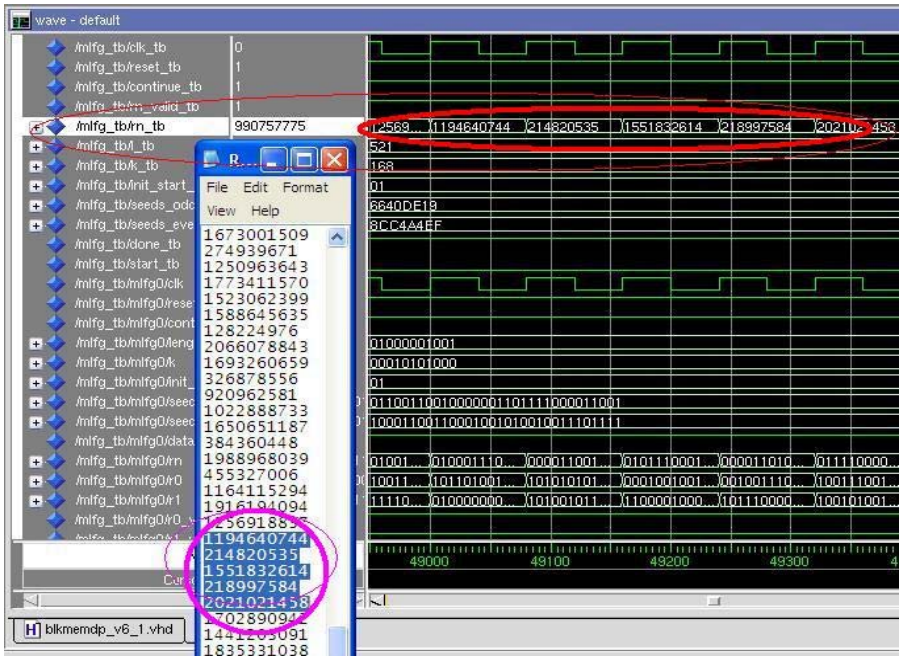

Figure 10. ModleSim Simulation Results – {521,168}

5.3 Synthesis Results

The implementation was synthesized with Synplicity Synplify Pro targeting on several Xilinx FPGA devices including Virtex-II Pro XC2VP50, Virtex-4 XC4VLX80 and Spartan-3 XC3S200 FPGAs. The Virtex-II Pro XC2VP50 contains 23,616 slices that each contains two storage elements (register bit) and two function generators (Look-Up Tables). The Virtex-4 XC4VLX80 holds 35,840 slices. The Sparten-3 XC3S200 has 1920 slices. The implementation results shown in the Table 8 demonstrates that the PALFG core is general and portable for a range of FPGA devices. Note that the resource usage (in LUT) is quite modest, demanding a miniscule fraction per generator(2%) when implemented on FPGAs such as the Xilinx Virtex II Pro VP50 devices contained in the Cray XD-1. Furthermore, from the table, this implementation yields high performance. The maximum generation rate reaches 206MHz on Xilinx Virtex-4 chips.

Table 8. PALFG Source Usage and Performance

FPGA Devices.	Slices	Usage	Rate (MRNS)
Virtex-II Pro XC2VP50	1100	2%	162.0
Virtex-4 XC4VLX80	1100	1.5%	206.0
Spartan-3 XC3S200	1100	57%	105.0

5.4 Post Synthesis Simulation

After synthesis, the design is again simulated. This post-synthesis simulation also verifies that the design is translated correctly into a netlist. The post-synthesis simulation timing results are more accurate than pre-synthesis as it includes logic delay information. Figure 11 is one example. This random number generator is the {607, 334} PALFG. Comparing the golden results with for parameter sets of random number generators, the synthesized design is verified.

5.5 Placement and Routing (PAR)

Placement and Routing follows synthesis and simulation. PAR involves the process of interconnecting other primitives within the slice to a matrix of wire segments, programmable switches and routing resources within FPGAs. This implementation has been placed and routed by the Xilinx ISE tool, fitting into the Xilinx VirtexII Pro XC2VP50 device.

During the implement design procedure, the translate process first merges all of the input netlists and design and outputs a Xilinx native generic database (NGD) file. Secondly, the map process maps the logic defined by an NGD file into FPGA elements.

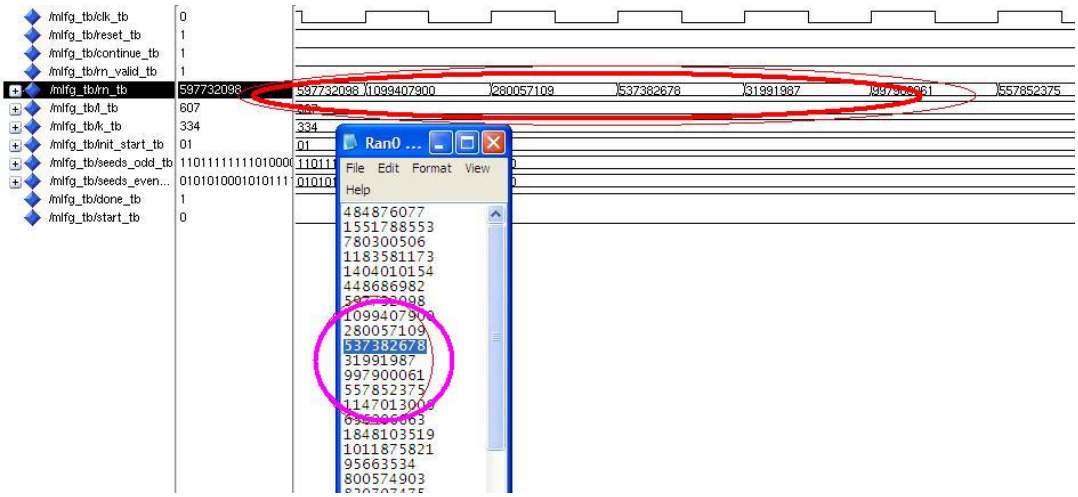

Figure 11. Post-Simulation –{607, 334}

The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA. Finally the place and route process takes a mapped NCD file for bitstream generation. The final report in ISE demonstrates that all signals have been completely routed

5.6 FPGA Implementations

SoC integrates all components of an electronic system into a single chip. Through the bus from the embedded PowerPC microprocessor core on chip, all components are connected and data can be routed directly between external interfaces and memory. Figure 12 shows the test platform. Through the bus, the parameter sets and seeds will be transferred by the PowerPC microprocessor (IBM PPC405) and the results will be read from addressable memory, forwarded to the RSR232, and finally shown on the host PC hyper-terminal window so that the random numbers can been seen and made available for testing.

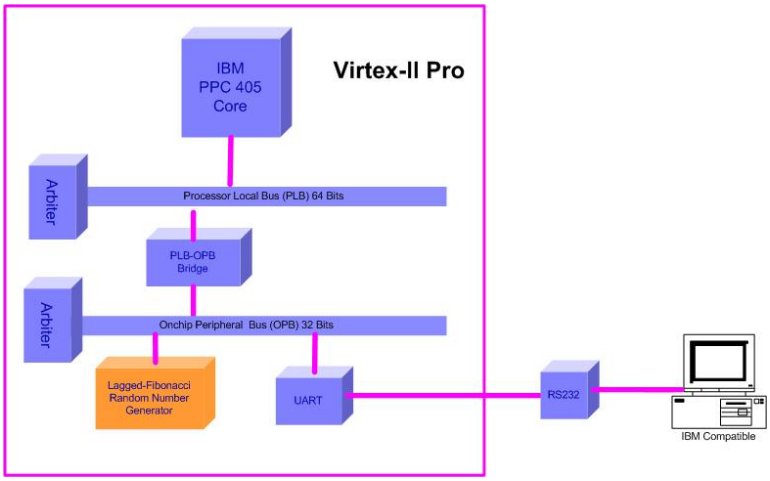

Figure 12. FPGA Implementation Test Platform

The Xilinx University Program Board (XUP) was used in this FPGA implementation verification. XUP board has one Xilinx VirtexII Pro XC2VP30 device, which contains 13,696 slices. Two PPC405 cores are on the chip. In this implementation, only one PPC405 core is used, with its frequency configured as 300MHz. Both UART module and PALFG module are attached to the OnChip peripheral bus (OPB), whose frequency is 100MHz. Given the general PALFG interface scheme (in Chapter 3), three addressable registers and two 8KB on chip memory units are included as the temporary memory to buffer the initial seeds or store the temporary results from the PLAFG core. C code was compiled for the PPC405 to execute initialization, generation and read results of three sequences.Following four implementation steps: synthesis, placement and routing, generating bitstream, and downloading, the results are finally shown on the Windows Hyperterminal software. All the 11 SPRNG PALFG parameter sets have been tested and confirmed. The Figure 13 shows one example of the results. The experiment results matches the golden results for each configuration. These tests demonstrate the design and implementation of PALFG is correct.

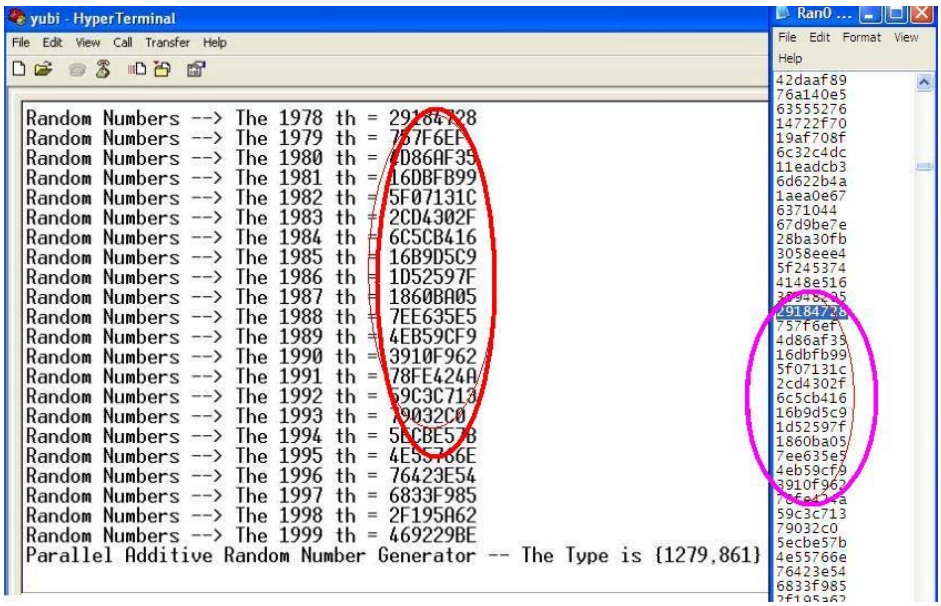

Figure 13. FPGA Implementation Verification Results – {1279, 861}

5.7 Performance Summary

As verified for both RTL and FPGA implementation, the PALFG emulates the SPRNG PALFG software correctly. The goal of the hardware PALFG implementation is to speed up the performance. Therefore, based on the synthesis results and SPRNG software execution performance, in Table 9, timing benchmarks for the SPRNG PALFG software generator of various processor architectures are reported [1]. Results for all Intel architectures are remarkably similar and produce around 50 million random numbers per second (MRNS). Among all the processors, the AMD Opteron270 presents the best performance, providing 83.3 MRNS. Comparing the 162 MRNS of our present Virtex II Pro hardware implementation to the best performance of the Opteron 270, the design provides about 2 times the throughput despite running at a clock speed 10 times slower.

Table 9. PALFG Timing Performance Comparisons[31]

Processor	Clock	Rate (MRNS)
Virtex-II Pro FPGA	162 MHz	162.0
Cray X1E	400 MHz	2.9
Intel Itanium2	1.5 GHz	45.7
IBM Power4+	1.7 GHz	29.4
AMD Opteron 270	2.0 GHz	83.3
Intel Xeon	2.4 GHz	56.9
Intel Pentium D	2.8 GHz	52.6

In terms of efficiency, the implementation appears to be nearly 20 times more efficient than the fastest software implementations. As the resources usage of the implementations is small, users can easily boost the efficiency another order of magnitude by placing multiple generators on one FPGA chip.

6. SPRNG Core Targeted on Supercomputer Cray XD1

As the PALFG development targets computational science on engineering applications on the supercomputers, this chapter illustrates how to apply the PALFG core to the Cray XD1. In the first two sections, the Cray XD1 architecture and necessary IP cores are introduced. Then following the design template proposed by [1], the implementation work of PALFG is specified in details. Simulation results and part of synthesis results are shown in the last section.

6.1 Cray XD1 Architecture

The basic architecture unit of the Cray XD1 system is the Cray XD1 chassis. One chassis contains twelve 64-bit AMD Opteron processors, twelve rapid array processors, six application acceleration processors (FPGA) and a management processor[1]. In each chassis, there are one to six compute blades. Each blade includes two 64-bit Opteron processors, DDR SDRAM, a RapidArray processor and a connector for an expansion module. The expansion module is the main module that fulfills the supercomputing reconfigurable functions. The expansion module has an additional RapidArray processor, one FPGA, four QDRII SRAMs for the FPGA and a programmable clock source for the FPGA as shown in Figure 14. The RapidArray processor provides the interface for the FPGA to connect to the local Opteron processors. The QDR SRAMs provide high-speed storage for the FPGA. The programmable clock enables the user to set the speed of the FPGA. On the current Cray XD1, Xilinx VirtexII pro XC2VP50 FPGA is applied as the application acceleration processor(AAP). XC2VP50 contains 53136 logic cells and 232 18Kb Block RAMS, which is high density and high performance system devices. Due to its characteristics, users can program computationally intensive algorithms on the FPGA.

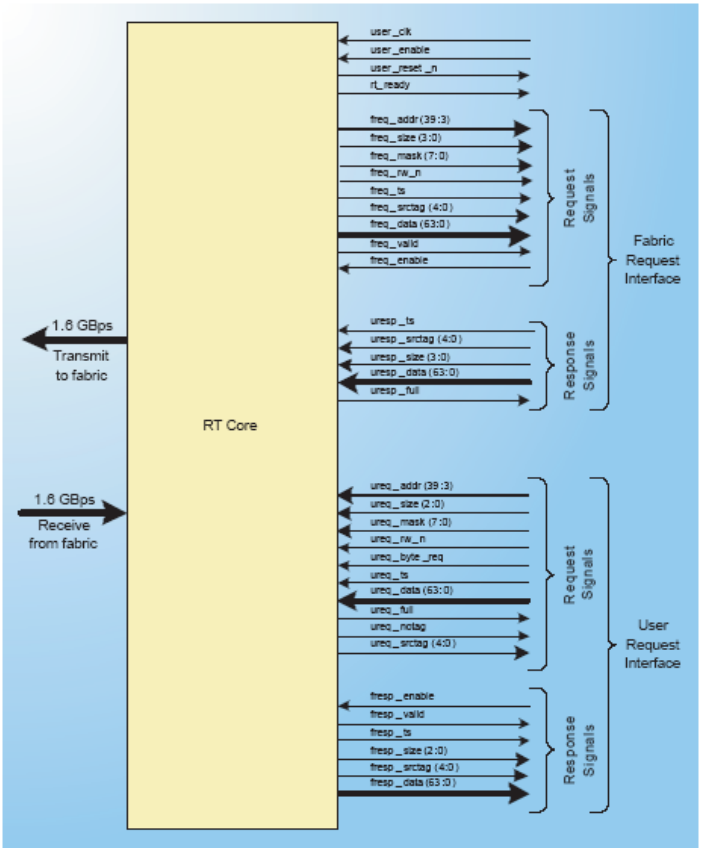

Figure 14. RapidArray Transport Module Interface [33]

6.2 Cray RapidArray Transport IP Core

The RapidArray Transport core interface, which provides the RapidArray fabric interface to an FPGA design, plays an extremely important role in the Cray XD1 FPGA development. The functionality of this IP core is to initiate and process responses for read and write transactions across the fabric. Two interfaces are given to this core: fabric request and user request. The fabric request interface issues read and write requests to the user logic. The user request processes read and write requests from the user logic. The maximum speed of the RT interface reaches 200 MHz. Figure 14 illustrates the Cray XD1 RT Interface.

6.3 Cray XD1 Design Template

The Cray XD1 development process follows the general FPGA design flows: HDL, Synthesis, Simulation and Implementation. After implementation, the final output is an FPGA binary file that contains the configuration bit stream for a specific Xilinx device. The file is then converted to a Cray proprietary format file (design.bin.ufp). Finally, the new created file can be downloaded to the FPGA AAP.

The Cray FPGA Development Document [33] provides a design template structure for the users to follow as show in Figure 15. The top-level VHDL file forms a wrapper that combines several logic components. Two of the components are the required logic cores for connecting to the RapidArray processor and to the QDRII SRAMs. A third component generates the user-programmable clock. The fourth component is the user application component that can be written by users. This template provides a flexible and easy method for users to port their IP cores to the Cray XD1. The primary requirement is that the user logic component should have the required interface signals to the others.

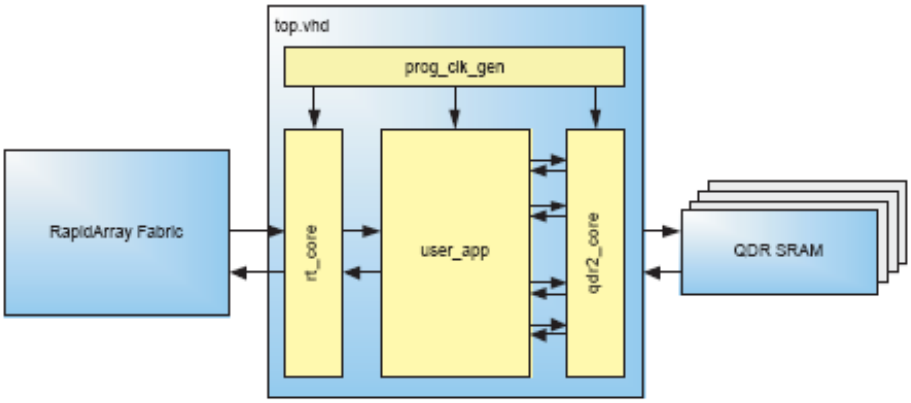

Figure 15. FPGA Organization in Cray XD1 [33]

6.4 SPRNG Interface Implementation Targeted on Cray XD1

Following the design template structure, we placed the SPRNG core and the interfaces in the user_app block. Since the SPRNG core has been already generated, the work on Cray XD1 is to develop a user interface and provide the required interface signals. As mentioned in the previous chapters, to generate random number, the PALFG core needs to accept parameter sets and seed values. In the Cray XD1 application, these values are transferred through RappidArray processor and need to be stored in the buffer. Thus how to instantiate the RT core interface, how to make internal memory addressable, and how to interface to the PALFG module are essential issues in this development. Since QDR is not necessary in this application, the QDR support is excluded.

6.4.1 Operational Scenario

To the node processor, the FPGA appears to be mapped in its virtual address space. Registers and BRAMs in the FPGA both have a corresponding virtual address space through which the node processor can be addressed. When the node processor accesses the

address space allocated for FPGA, the requests will be sent through the HyperTransport bus and issued to the RapidArray processor. Then the RapidArray processor will send the requests to the FPGA through the RapidArray Transport (RT) Interface IP core. Given different virtual addresses, the requests may be forwarded to the registers that set the control signals or parameter sets of SPRNG to the PALFG. The requests may also be sent to the internal FPGA BRAMs to initialize the seed arrays or read the random numbers from them. To the accelerated FPGA, after the FPGA receives the requests from the RT, it will acknowledge the request through the RT. Finally the node processor will receive the response from the HyperTransport bus and continue its execution. Figure 16 shows the logical view of this scenario.

6.4.2 User Application Block

The user block contains four sub blocks: the RT Client block, Register Interface block, RAM Interface block, and the PALFG block (Figure 17). The RT client block is an instance of the RT core and is responsible for acknowledging a fabric request, forwarding the request to the correct sub-block, and polling any required response.

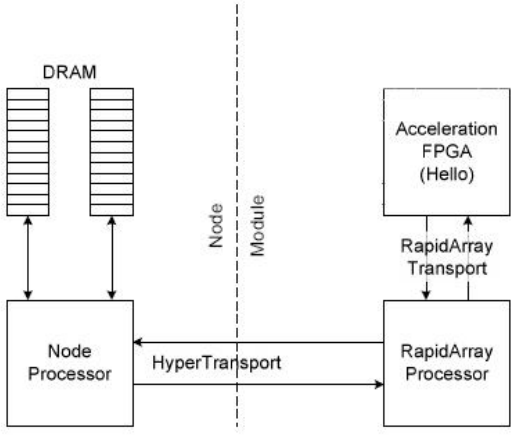

Figure 16. Operational Scenario in Cray XD1 [34]

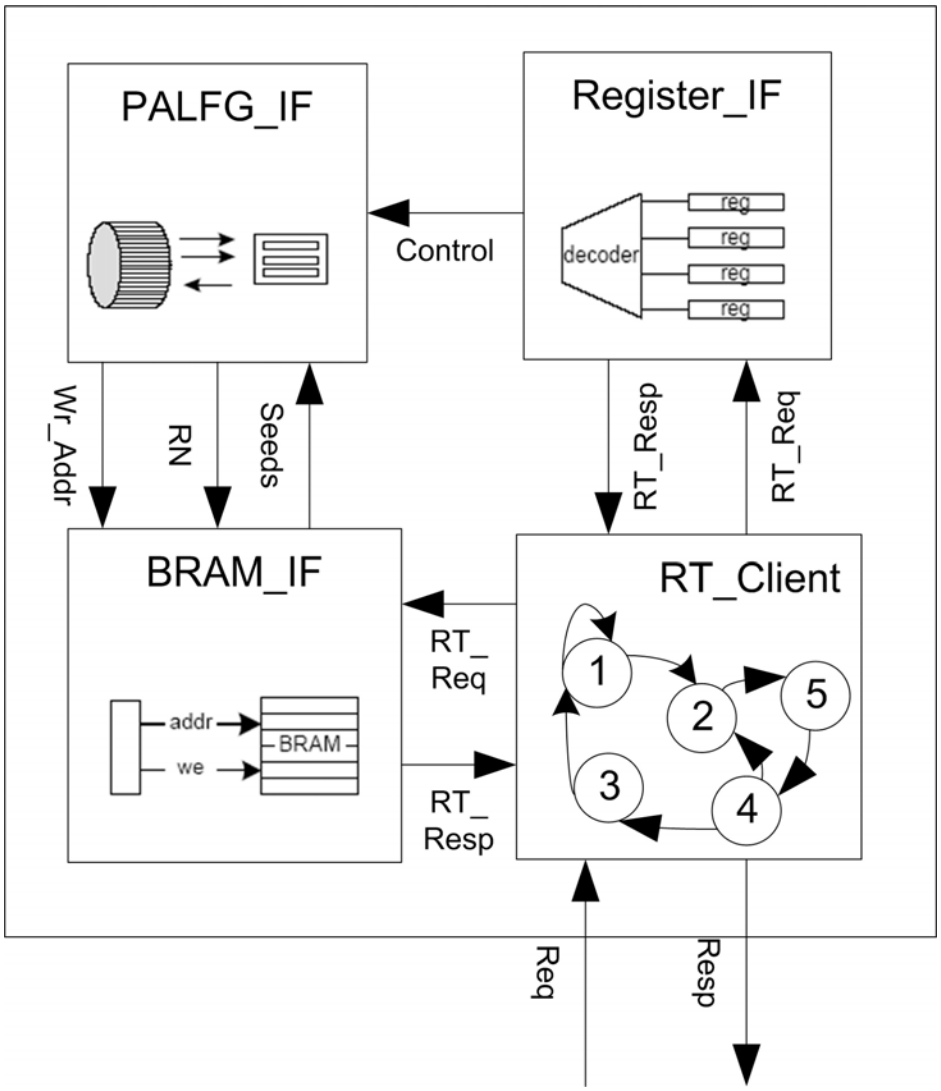

Figure 17. The Application Block Shceme

To fulfill the functionality, the RT client block executes a large state machine that controls the responses to the fabric request interface. The RT client has 5 states: idle, rd_stall, wr_stall, wr and rd. Figure 18 illustrates the state transform [34]. The register interface block provides a set of writable processor interface registers. The registers are used to transfer the control commands and parameter sets from the RappidArray processor. These registers are addressable so that the programmer can control the PALFG by setting the values in register. The Block RAM Interface block (Figure 19) provides read and write access for a processor to 16 Kbytes of internal Xilinx Block RAM. The software program running on the Opteron processor can write the seeds values directly into the block RAM and read the random number results from it. The Block RAM Interface consists of some simple RAM control logic and a set of eight Xilinx Block RAMs. Figure 20 shows the logic block diagram[2].

6.4.3 Memory Mapping

The accelerated FPGAs' address region is 128 MB in the processors virtual address space. In this application, one 16 KB BRAM and three registers in the FPGA are mapped as the following virtual address so that the host node can access either of them (Table 10).

6.4.4 RTL Simulation

The application is simulated at a functional level using a test bench in VHDL. The test bench instantiates the PALFG_IF, BRAM_IF and Reg_IF three block modules. First the test module writes seeds, parameter sets and signals separately to the BRAM_IF and Reg_IF blocks. Then through the control signals from REG_IF block, the PALFG_IF start to generate random numbers and write results back to the BRAM_IF block. The simulation results verify that the identical random numbers to SPRNG are written to the memory that can be accessed by host node. Due to the synthesis debug issues and limited time, the implementation of PALFG on Cray XD1 is not complete.

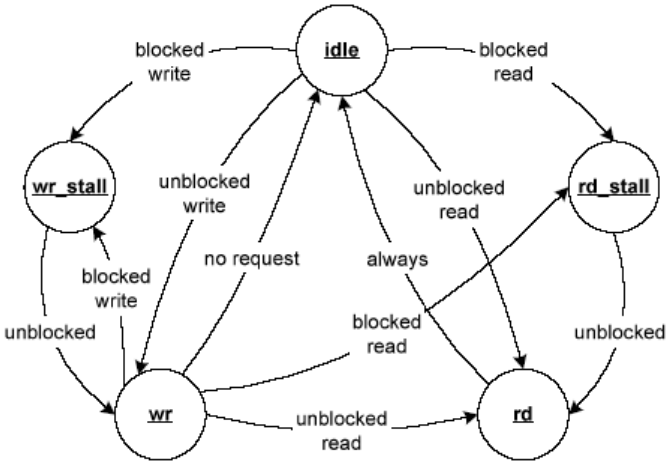

Figure 18. RapidArray Transport Client State Machine [34]

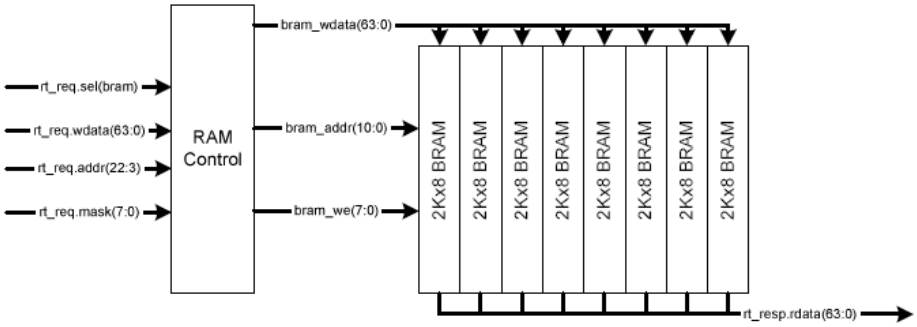

Figure 19. Block RAM Interface Block Diagram [34]

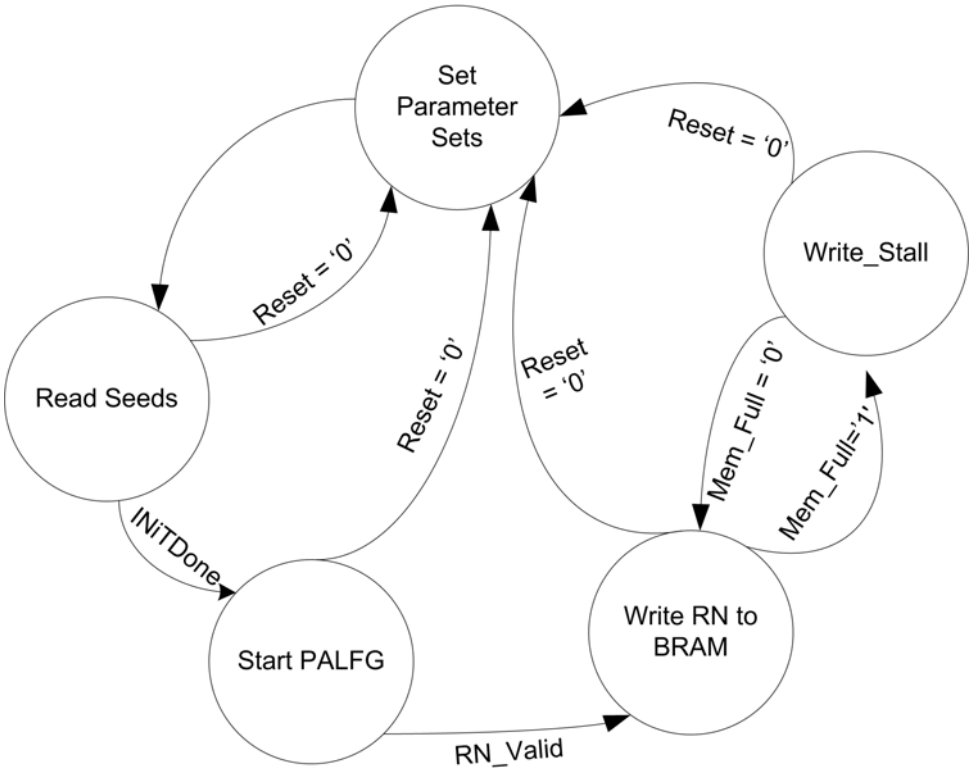

Figure 20. PALFG Interface Block Diagram

Table 10. XUP Memory Mapping

Offset	Size (Bytes)	Region Type	Description
0x00000000- 0x00003FFF	16K	Block RAM	Internal Xilinx Block BRAM
0x40000010	8	Register 1	User State Control Command
0x40000018	8	Register 2	Set Parameter L
0x40000020	8	Register 3	Set Parameter K

7. Conclusion

This work leads the way to the first hardware-based implementation of SPRNG for the scientific computing speedup purpose. This implementation provides identical results to the SPRNG's PALFG and provides the flexibility in the user environments. Due to its fast generation speed and friendly interface for users, this uniform random number generator is being targeted as an open core for parallel scientific computing.

7.1 Summary

There is always a clear and urgent need for speedup in parallel scientific computing. Due to the occurrence of reconfigurable supercomputing, hardware and software co-implementation in supercomputing attracts more and more attention in this parallel scientific computing area. A FPGA design and implementation of the SPRNG's PALFG is presented with consideration of a number of tradeoffs: high throughput and flexibility.

First, given different cases of the SPRNG's PALFG, two flexible designs were developed for all the parameter sets of PALFG: Odd-Even PALFG and Odd-Odd PALFG. These two designs implement the PALFG algorithm by setting the parameter sets and each of the designs includes two ALFG modules, controller blocks and execution modules. The ALFG modules contain three basic blocks -- RAM arrays. The RAM arrays are implemented to be the circular buffers. Each RAM array is composed of four 9-bit by 2048-bit BRAMs so that it makes the architecture support all the parameter sets of the PALFG only if l is less than 2048. Furthermore, a friendly interface architecture was presented for users to combine the PALFG module with other external devices.

Second, verification was performed throughout all the design. Each component in the PALFG was individually tested before being integrated together into PALFG. A prototype was implemented in software. Comparison with the golden results that the prototype

produces, the PALFG module was verified using simulation and FPGA implementation. The implementation passed all the verification tests with identical results to SPRNG.

Third, the PALFG implementation was targeted on the Cray XD1 supercomputer. A general user block for PALFG applied into Cray XD1 is presented. A user block in the FPGA for Cray XD1 was implemented using VHDL. This block contains RappidArray Transport client block, Register interface block, RAM interface block and the PALFG block. This user block passed the RTL simulation. Due to the time limitation, the final PALFG implementation hasn't been executed on CrayXD1.

7.2 Future Work

As mentioned above that the final PALFG Cray XD1 implementation is not complete, so the first future work is to make the PALFG work on the Cray XD1. Since the architecture, the VHDL codes and C codes for Cray XD1 are complete, it won't take long time to fulfill this work in the future.

Several types of random number generators are provided in the SPRNG library such as combined Multiple Recursive Generator, 48-bit and 64-bit linear congruential generators, and Prime Modulus Linear Congruential Generator. In this thesis, only the PALFG was designed and implemented as the first try to verify the feasibility of the SPRNG library in hardware implementation. So in the future, more parallel random number generators in SPRNG can be implemented for more parallel computing applications.

A general user block for applying PALFG into Cray XD1 is presented. More efficient modules can be adapted to this user block such as double buffering to increase the execution speed of the user block to reach the maximum speed of the PALFG.

As the Gaussian distributed random generator is also widely used in parallel scientific computing, we also can try to implement a Gaussian distributed random number generator based on the PALFG's uniform random number generator.

List of References

[1] Michael Mascagni, Ashok Srinivasan, “Algorithm 806: SPRNG: a scalable library for pseudorandom number generation”, *ACM Transactions on Mathematical Software (TOMS)*,Vol 26 Issue 3, Sep 2000

[2] Simka, M; Drutarovsky, M; Fischer, V; Fayolle,J “ Model of a true random number generator aimed at cryptographic applications”, *Circuits and Systems*, 2006

[3] N. C. Metropolis and S. M. Ulam, “The Monte-Carlo method,” *J. Am.Stat. Assoc*., vol. 44, p. 335, 1949.

[4] H. Zaidi and G. Sgouros, Eds., “Therapeutic Applications of Monte CarloCalculations in Nuclear Medicine”, *ser. Series in Medical Physics and Biomedical Engineering. Institute of Physics*, 2002, vol. 24.

[5] P. Glasserman, “Monte Carlo Methods in Financial Engineering”, *ser.Stochastic Modelling and Applied Probability*. Springer, 2003, vol. 53.

[6] Melissa C.Smith and Gregory D.Peterson, “Parallel Application Performance on Shared High Performance Reconfigurable Computing Resources”, *Performance Evaluation*. 60(1-4): 107-125, 2005

[7] Junqing Sun, Gregory D. Peterson. "Effective Execution Time Estimation for Heterogeneous Parallel Computing" *Parallel and Distributed Computing Systems*, San Francisco, California, USA, Sep, 2006

[8] Maya B and Pau. S.graham, “Reconfigurable Supercomputing”, *Springer*, 2005

[9] D.Lund, V.Barekos, and B.Honary, “Convolutional decoding for reconfigurable mobile systems”, *IEE Conference Publications* (2001), 297 –301

[10] M.Kim, K.Ichige, and H.Arai, “Design of Jacobi EVD processor based on CORDIC for DOA estimation with MUSIC algorithm”, *IEICE Transactions on Communications* E85/B (2002;2003), no.12, 2648 -55

[11]Scott.E Fields, “Hardware Design and Implementation of Role-based Cryptography”, The University of Tennessee, Knoxville, Dec 2005

[12] Alirez Hodjat and Ingrid Verbauwhede, "A 21.54 gbits/s fully pipelined AES processor on FPGA", *Preceedings of IEEE Symposium on FPGAs for custom computing machines* (Napa, CA) (J.M.Arnold and K.L.Pocek, eds), April 2004

[13] Tripp, J.L; hanson, A.A; Gokhale,M; Mortveit, H, "Partitioning Hardware and Software for Reconfigurable supercomputing applications: A case study ", Supercomputing 2005, *Proceedings of the ACM/IEEE SC2005 Conference* 12-18, Nov.2005 27-27

[14] Barrett, C. L., Beckman, R. J., Berkbigler, K. P., Bisset, K. R., Bush, B. W., Campbell, K., Eubank, S., Henson,K. M., Hurford, J. M., Kubicek, D. A., Marathe, M. V., Romero, P. R., Smith, J. P., Smith, L. L., Stretz, P. E., Thayer, G. L., " Transportation Analysis Simulation System (TRANSIMS)", portland study reports.

[15] Gregory Dean Peterson, "Parallel Application Performance on Shared, Heterogeneous Workstations", Washington University, St.Louis, MO, 1994

[16] [online] http://en.wikipedia.org/wiki/Linear_feedback_shift_register

[17] Peter Martin, "An Analysis of Random Number Generators for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C ", *J.Genetic Programming and Evolvable Machines*, 317-343, Dec. 2001

[18] Jason Stredwick, "Random Number Generators In Use", MSU 2004, [online] http://www.jasonstredwick.net/Data/rng.ppt

[19] K.H.Tsoi, K.H.Leung and P.H.W.Leong, " Compact FPGA-based True and pseudorandom number generators", Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custon Computing Machines, 2003

[20] Matti tommiska and Jarkko Vuori, "Hardware implementation of GA", Proceedings of the Second Nordic Worshop on Genetic Algorithms and their applications (2NWGA), Vaasa, Finland, 1996

[21]Makoto Matsumto and Takuji Nishimura, " Mersenne Twister : A 623-dimensionally equidistributed uniform pseudorandom number generator, " ACM Transactions on Modeling and computer Simulation, vol 8, no.1 pp.3-30 January 1998

[22] "Cray XD1 Mersenne Twister Acceleraotr (MTA) FPGA design", Cray Inc 2005

[23] Alexander S.Pasciak and John R.Rord "A New High Speed Solution for the Evaluation of Monte Carlo Radiation Transport Computations", *Nuclear Science*, IEEE Transactions on Vol 53, Issue 2, April 2006

[24] M.Mascagni, Parallel Linear Congruenial Generators with Prime Moduli, Parallel Computing, 24(1998) 923-936

[25] Zhou,M; Mascagni, M. The cycle server: a Web platform for unning parallel Monte Carlo applications on a heterogenerous Condor pool of workstations", *Parallel Processing*, 2000 International Workshops on 21-24 Agu.2000 Pages: 111-118

[26] Dewaraja, .K; Ljungberg, M; Majumdar, A; Bose, A,; Koral, K.F; "A Parallel Monte Carlo code for planar and SPEC imageing: implementation, verification and applicationsin I SPECT", *Nuclear Science Symposium Conference Record*, 2000 IEEE Volume 3, 15-20 Oct 2000 Pages :20/30- 20/34

[27] [online] http://en.wikipedia.org/wiki/Diehard_tests

[28] [online] http://random.com.hr/products/random/Diehard.html

[29] Johnson B.C. "Radix-b extensions to some common empirical tests for pseudorandom number generators". *ACM Transactions on Modelling and Computer Simulation*, 6(4): 261-273, 1996

[30] Knuth D.E 1997 "The Art of Computer Programming", Volume 2 Seminumerical Algorithms. 3rd ed. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

[31] Yu Bi, Gregory.D.Peterson, G.Lee.Warren, Robert J.Harrison, "A Reconfigurable Supercomputing Library for Accelerated Parallel Lagged-Fibonacci Pseudorandom Number Generation", *Supercomputing Conference*, Tempa, 2006

[32] Yu.Bi, Gregory.D.Peterson, G.L.Warren, and R.J.Harrison, "Hardware acceleration of parallel lagged-Fibonacci pseudo random number generation," *Intl Conf. on Engineering of Reconfigurable Systems and Algorithms*. June, 2006

[33] Cray Inc. "Cray XD1 FPGA Development"

[34] Cray Inc. "Cray XD1 Hello World FPGA Design"

Appendices

Appendix I-- Main PALFG VHDL Codes

1. PALFG Top Level VHDL Codes—K= EVEN

```
/************************************************************
Name of code:    MLFG.VHD

Purpose of code:    Modified PLFG K=Even

Author of code:    Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in
the Tennessee Advanced Computing Laboratory.

Copyright (C) 2006 Yu Bi and Gregory D. Peterson

This library is free software; you can redistribute it and/or modify it under the terms of
the GNU Lesser General Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any later version. This library is
distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License along with
this library; if not, write to
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA    02110-1301 USA.

You may also view the GNU Lesser General Public License at
http://www.gnu.org/licenses/lgpl.html

For additional information or queries, send email to gdp@utk.edu.
************************************************************/


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
```

```
entity MLFG is

        port( clk    : in std_logic;
          reset: in std_logic;
              MLFGcontinue: in std_logic;
          length: in std_logic_vector( 10 downto 0); -- for example 31, 5,    length=30;
          k: in std_logic_vector( 10 downto 0);
          Init_Start: in std_logic_vector( 1 downto 0);
          Seeds_odd: in std_logic_vector( 31 downto 0);
          Seeds_even: in std_logic_vector( 31 downto 0);
          Data_valid: out std_logic;
                RN: out std_logic_vector(31 downto 0)
                );
end MLFG;

architecture a of MLFG is

signal R0 : std_logic_vector(31 downto 0);
signal R1 : std_logic_vector(31 downto 0);
signal R0_valid: std_logic;
signal R1_valid: std_logic;
signal sRN: std_logic_vector(31 downto 0);
signal state: std_logic_vector( 1 downto 0);

component MLFGcore

        port( clk    : in std_logic;
          reset: in std_logic;
                continue: in std_logic;
          length: in std_logic_vector(10 downto 0);
          k: in std_logic_vector( 10 downto 0);
          Init_Start: in std_logic_vector( 1 downto 0);
          Seeds: in std_logic_vector( 31 downto 0);
          Data_valid: out std_logic;
                Data: out std_logic_vector(31 downto 0)
                );
end component;
```

```
component MLFGExecute

            port( clk : in std_logic;
              Adder_odd: in std_logic_vector(31 downto 0);
                    Adder_even: in std_logic_vector(31 downto 0);
                    RN: out std_logic_vector(31 downto 0)
                 );
end component;



begin

MLFG_even: MLFGcore port map
          (
              clk => clk,
              reset => reset,
                    continue => MLFGcontinue,
              length => length,
              k => k,
              Init_Start => Init_Start,
              Seeds => Seeds_even,
              Data_valid => R0_valid,
                    Data => R0
                    );

MLFG_odd: MLFGcore port map
           (
              clk => clk,
              reset => reset,
                    continue => MLFGcontinue,
              length => length,
              k => k,
              Init_Start => Init_Start,
              Seeds => seeds_odd,
              Data_valid => R1_valid,
                    Data => R1
                    );
MLFG_Execute: MLFGExecute port map
```

```
            (
        clk => clk,
        Adder_odd => R1,
              Adder_even => R0,
              RN => sRN
           );


process(clk)
 begin
     if reset = '0' then
          state<="00";
          data_valid <='0';
     else
          if clk' event and clk ='1' then

               if MLFGcontinue = '1' then
                    case state is
                        when "00" =>
                             if (R0_valid = '1') then
                                  if(R1_valid = '1') then
                                       state <="01";
                                  end if;
                             else
                                  state <="00";
                             end if;
                        when "01" =>
                             RN(30 downto 0)<= sRN(31 downto 1);
                             state <="10";
                        when "10" =>
                              Data_valid <='1';
                              RN(30 downto 0) <=sRN(31 downto 1);
                        when others =>

                             null;
                   end case;


               end if;
```

```
            end if;
        end if;
end process;
RN(31) <= '0';
end;
```

2. PALFG Kernel VHDL Codes – K= EVEN

```
/************************************************************
Name of code:    MLFGcore.VHD

Purpose of code:    Modified PLFG kernel core K=Even

Author of code:    Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in
the Tennessee Advanced Computing Laboratory.

Copyright (C) 2006 Yu Bi and Gregory D. Peterson

This library is free software; you can redistribute it and/or modify it under the terms of
the GNU Lesser General Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any later version. This library is
distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License along with
this library; if not, write to
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA    02110-1301 USA.

You may also view the GNU Lesser General Public License at
http://www.gnu.org/licenses/lgpl.html

For additional information or queries, send email to gdp@utk.edu.
************************************************************/

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
```

```
entity MLFGcore is

            port( clk    : in std_logic;
              reset: in std_logic;
                    continue: in std_logic;
              length: in std_logic_vector ( 10 downto 0); --    for example 31, 5 length =
30;
              k: in std_logic_vector(10 downto 0);
              Init_Start: in std_logic_vector( 1 downto 0);
              Seeds: in std_logic_vector(31 downto 0);
              Data_valid: out std_logic;
                    Data: out std_logic_vector(31 downto 0)
                    );
end MLFGcore;

architecture a of MLFGcore is
signal Raddr_1 : std_logic_vector( 10 downto 0);
signal Raddr_2: std_logic_vector( 10 downto 0);
signal Raddr_3: std_logic_vector( 10 downto 0);
signal Waddr_1: std_logic_vector( 10 downto 0);
signal Waddr_2: std_logic_vector( 10 downto 0);
signal tempRaddr_1: std_logic_vector( 10 downto 0);
signal tempRaddr_2: std_logic_vector( 10 downto 0);


signal Rdata_1: std_logic_vector( 31 downto 0);
signal Rdata_2: std_logic_vector( 31 downto 0);
signal Rdata_3: std_logic_vector( 31 downto 0);
signal Wdata_1: std_logic_vector( 31 downto 0);
signal Wdata_2: std_logic_vector( 31 downto 0);


signal Raddr_en : std_logic;

signal Initi_Done: std_logic;
signal initivalue_done: std_logic;
signal initvalue_1:std_logic_vector(10 downto 0);
signal initvalue_2: std_logic_vector(10 downto 0);
signal initvalue_3: std_logic_vector(10 downto 0);
```

```
signal halflength: std_logic_vector(10 downto 0);

signal Data_in_1: std_logic_vector(31 downto 0);
signal Data_in_2 : std_logic_vector( 31 downto 0);

signal state: std_logic_vector( 2 downto 0);
signal counterlength : std_logic_vector( 10 downto 0);
component First is
    port(           clk    : in std_logic;
                reset: in std_logic;
                        continue: in std_logic;
                Init_Start: in std_logic_vector ( 1 downto 0);
                Raddr_1: in std_logic_vector(10 downto 0);
                Raddr_2: in std_logic_vector(10 downto 0);
                Raddr_3: in std_logic_vector(10 downto 0);
                Waddr_1: in std_logic_vector(10 downto 0);
                Waddr_2: in std_logic_vector(10 downto 0);
                Data_in_1: in std_logic_vector(31 downto 0);
                Data_in_2: in std_logic_vector(31 downto 0);
                        Data_out_1: out std_logic_vector(31 downto 0);
                Data_out_2: out std_logic_vector(31 downto 0);
                Data_out_3: out std_logic_vector(31 downto 0);
                Initi_Done: out std_logic;
                initivalue_done: out std_logic

                        );
end component;

component Second is
     port(

                clk: in std_logic;
                Adder_1: in std_logic_vector( 31 downto 0);
                        Adder_2: in std_logic_vector(31 downto 0);
                Adder_3: in std_logic_vector(31 downto 0);
                        Sum_1: out std_logic_vector(31 downto 0);
                Sum_2: out std_logic_vector(31 downto 0)
                   );
end component;
```

```
component SetRaddr
      port( clk : in std_logic;
                reset: in std_logic;
                  initvalue_1: in std_logic_vector(10 downto 0);
                initvalue_2: in std_logic_vector(10 downto 0);
                initvalue_3: in std_logic_vector(10 downto 0);
                length: in std_logic_vector(10 downto 0);
                Raddr_1_en: in std_logic;
                Raddr_2_en: in std_logic;
                Raddr_3_en: in std_logic;
                Raddr_1: out std_logic_vector(10 downto 0);
                Raddr_2: out std_logic_vector(10 downto 0);
                Raddr_3: out std_logic_vector(10 downto 0)
                      );
end component;

component SetWrite

              port( clk    : in std_logic;
                Raddr_1: in std_logic_vector(10 downto 0);
                Raddr_2: in std_logic_vector(10 downto 0);
                Waddr_1: out std_logic_vector(10 downto 0);
                Waddr_2: out std_logic_vector(10 downto 0)
               );
end component;


begin
FirstStep: First
port map(
               clk => clk,
                reset => reset,
                        continue => continue,
                Init_Start => Init_Start,
                Raddr_1 => Raddr_1,
```

```
                Raddr_2 => Raddr_2,
                Raddr_3 => Raddr_3,
                Waddr_1 => Waddr_1,
                Waddr_2 => Waddr_2,
                Data_in_1 => Data_in_1,
                Data_in_2 => Data_in_2,
                 Data_out_1 => Rdata_1,
                Data_out_2 => Rdata_2,
                Data_out_3 => Rdata_3,
                Initi_Done => Initi_Done,
                initivalue_done =>initivalue_done
                        );


SecondStep: Second
      port map(
                clk => clk,
                Adder_1 => Rdata_1,
                        Adder_2 => Rdata_2,
                Adder_3 => Rdata_3,
                        Sum_1 => Wdata_2,
                Sum_2 => Wdata_1
                    );



ReadAdrr: SetRaddr
      port map( clk => clk,
                reset => reset,
                initvalue_1=>initvalue_1,
                initvalue_2 => initvalue_2,
                initvalue_3=>initvalue_3,
                length => counterlength,
                Raddr_1_en => Raddr_en,
                Raddr_2_en => Raddr_en,
                Raddr_3_en => Raddr_en,
                Raddr_1 => Raddr_1,
                Raddr_2 => Raddr_2,
```

```
                Raddr_3 => Raddr_3
                    );


WriteAddr: SetWrite

            port map ( clk => clk,
                Raddr_1 => tempRaddr_1,
                    Raddr_2 => tempRaddr_2,
                    Waddr_1 => Waddr_1,
                    Waddr_2 => Waddr_2
             );

WriteAddrDelay1clk: SetWrite
    port map ( clk => clk,
                Raddr_1 => Raddr_1,
                Raddr_2 => Raddr_2,
                Waddr_1 => tempRaddr_2,
                Waddr_2 => tempRaddr_1
                );

Data_in_1 <= Seeds when Init_Start = "00"
                else Wdata_1;
Data_in_2 <= Seeds when Init_Start = "00"
                else Wdata_2;


Data <= Wdata_2;

halflength(10) <= '0';

halflength(9 downto 0) <= length(10 downto 1);
counterlength <= length - 1 when initivalue_Done = '1'
            else "00000000000";

process (clk, reset)
begin
    if reset = '0' then
        state<="000";
```

```
        Raddr_en    <= '0';
                    Data_valid <= '0';
else
        if clk' event and clk ='1' then

            if continue = '1' then
              case state is
                    when "000" =>
                        if initivalue_Done = '1' then
                            state <="001";
                            initvalue_1 <= (others =>'0');
                            initvalue_2 <= length-k;
                            if halflength > k then
                                initvalue_3 <= length-k-k;
                            else
                                initvalue_3 <= length+length-k-k;
                            end if;

                        else

                            state <="000";
                        end if;
                    when "001" =>
                        if Initi_Done = '1' then
                            state <="010";
                        else
                            state <="001";
                        end if;
                    when "010" =>
                        Raddr_en <= '1';
                        state <="011";

                    when "011" =>
                        state <= "100";

                    when "100" =>

                        state <= "100";
                        Data_valid <='1';
```

```
                    when others =>
                        null;

                end case;
                 end if;
                end if;
      end if;
end process;
end;
```

3. PALFG Top Level VHDL Codes—K= ODD

```
/************************************************************
Name of code:    MLFG.VHD

Purpose of code:    Modified PLFG K=Odd

Author of code:    Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in
the Tennessee Advanced Computing Laboratory.

Copyright (C) 2006 Yu Bi and Gregory D. Peterson

This library is free software; you can redistribute it and/or modify it under the terms of
the GNU Lesser General Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any later version. This library is
distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License along with
this library; if not, write to
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA    02110-1301 USA.

You may also view the GNU Lesser General Public License at
http://www.gnu.org/licenses/lgpl.html

For additional information or queries, send email to gdp@utk.edu.
************************************************************/

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity MLFG is
```

```
            port( clk    : in std_logic;
              reset: in std_logic;
                  MLFGcontinue: in std_logic;
              length: in std_logic_vector( 10 downto 0); -- for example {31, 6},
length=31, k= 6;
              k: in std_logic_vector( 10 downto 0);
              Init_Start: in std_logic_vector( 1 downto 0);
              Seeds_odd: in std_logic_vector( 31 downto 0);
              Seeds_even: in std_logic_vector( 31 downto 0);
              Data_valid: out std_logic;
                  RN: out std_logic_vector(31 downto 0)
                    );
end MLFG;

architecture a of MLFG is

signal R0 : std_logic_vector(31 downto 0);
signal R1 : std_logic_vector(31 downto 0);
signal R0_valid: std_logic;
signal R1_valid: std_logic;
signal sRN: std_logic_vector(31 downto 0);
signal state: std_logic_vector( 1 downto 0);

component MLFGcore

            port( clk    : in std_logic;
              reset: in std_logic;
                    continue: in std_logic;
              length: in std_logic_vector(10 downto 0);
              k: in std_logic_vector( 10 downto 0);
              Init_Start: in std_logic_vector( 1 downto 0);
              Seeds: in std_logic_vector( 31 downto 0);
              Data_valid: out std_logic;
                    Data: out std_logic_vector(31 downto 0)
                    );
end component;

component MLFGExecute
```

```
        port( clk : in std_logic;
          Adder_odd: in std_logic_vector(31 downto 0);
                Adder_even: in std_logic_vector(31 downto 0);
                RN: out std_logic_vector(31 downto 0)
             );
end component;




begin

MLFG_even: MLFGcore port map
        (
          clk => clk,
          reset => reset,
                continue => MLFGcontinue,
          length => length,
          k => k,
          Init_Start => Init_Start,
          Seeds => Seeds_even,
          Data_valid => R0_valid,
                Data => R0
                );

MLFG_odd: MLFGcore port map
         (
          clk => clk,
          reset => reset,
                continue => MLFGcontinue,
          length => length,
          k => k,
          Init_Start => Init_Start,
          Seeds => seeds_odd,
          Data_valid => R1_valid,
                Data => R1
                );
```

```
MLFG_Execute: MLFGExecute port map
                    (
               clk => clk,
               Adder_odd => R1,
                     Adder_even => R0,
                     RN => sRN
                  );


process(clk)
 begin
     if reset = '0' then
          state<="00";
          data_valid <='0';
     else
          if clk' event and clk ='1' then

               if MLFGcontinue = '1' then
                     case state is
                        when "00" =>
                             if (R0_valid = '1') then
                                  if(R1_valid = '1') then
                                       state <="01";
                                  end if;
                             else
                                  state <="00";
                             end if;
                        when "01" =>
                             RN(30 downto 0)<= sRN(31 downto 1);
                             state <="10";
                        when "10" =>
                              Data_valid <='1';
                              RN(30 downto 0) <=sRN(31 downto 1);
                        when others =>

                             null;
                    end case;
```

```
                end if;
            end if;
        end if;
end process;
RN(31) <= '0';
end;
```

4. PALFG Kernel VHDL Codes – K=ODD

```
/************************************************************
Name of code:    MLFGcore.VHD

Purpose of code:    Modified PLFG kernel core K=Even

Author of code:    Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in
the Tennessee Advanced Computing Laboratory.

Copyright (C) 2006 Yu Bi and Gregory D. Peterson

This library is free software; you can redistribute it and/or modify it under the terms of
the GNU Lesser General Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any later version. This library is
distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License along with
this library; if not, write to
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA    02110-1301 USA.

You may also view the GNU Lesser General Public License at
http://www.gnu.org/licenses/lgpl.html

For additional information or queries, send email to gdp@utk.edu.
************************************************************/

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity MLFGcore is
```

```
            port(clk    : in std_logic;
              reset: in std_logic;
                  continue: in std_logic;
              length: in std_logic_vector ( 10 downto 0); --    for example 31, 5 length =
30;
              k: in std_logic_vector(10 downto 0);
              Init_Start: in std_logic_vector( 1 downto 0);
              Seeds: in std_logic_vector(31 downto 0);
              Data_valid: out std_logic;
                  Data: out std_logic_vector(31 downto 0)
                    );
end MLFGcore;

architecture a of MLFGcore is
signal Raddr_1 : std_logic_vector( 10 downto 0);
signal Raddr_2: std_logic_vector( 10 downto 0);
signal Raddr_3: std_logic_vector( 10 downto 0);
signal Waddr_1: std_logic_vector( 10 downto 0);

signal Waddr_3: std_logic_vector(10 downto 0);
signal tempRaddr_1: std_logic_vector( 10 downto 0);

signal tempRaddr_3: std_logic_vector( 10 downto 0);


signal Rdata_1: std_logic_vector( 31 downto 0);
signal Rdata_2: std_logic_vector( 31 downto 0);
signal Rdata_3: std_logic_vector( 31 downto 0);
signal Wdata_1: std_logic_vector( 31 downto 0);
signal Wdata_2: std_logic_vector( 31 downto 0);
signal Wdata_3: std_logic_vector( 31 downto 0);
signal Wdata_3_dly: std_logic_vector( 31 downto 0);


signal Raddr_1_en : std_logic;
signal Raddr_2_en : std_logic;
signal Raddr_3_en : std_logic;
signal Write_en_1: std_logic;
```

```
signal Write_en_3: std_logic;


signal Initi_Done: std_logic;
signal Initi_Done_1: std_logic;
signal Initi_Done_3: std_logic;

signal initivalue_done_1: std_logic;
signal initivalue_done_3: std_logic;
signal initivalue_done: std_logic;

signal initvalue_1:std_logic_vector(10 downto 0);
signal initvalue_2: std_logic_vector(10 downto 0);
signal initvalue_3: std_logic_vector(10 downto 0);


signal Data_in_1: std_logic_vector(31 downto 0);
signal Data_in_2 : std_logic_vector( 31 downto 0);

signal state: std_logic_vector( 2 downto 0);
signal counterlength : std_logic_vector( 10 downto 0);

component First is
    port(       clk    : in std_logic;
                reset: in std_logic;
                    continue: in std_logic;
                Init_Start: in std_logic_vector(1 downto 0);
                Raddr_1: in std_logic_vector(10 downto 0);
                Raddr_2: in std_logic_vector(10 downto 0);
                Waddr_1: in std_logic_vector(10 downto 0);
                Data_in_1: in std_logic_vector(31 downto 0);
           write_en : in std_logic;
                Data_out_1: out std_logic_vector(31 downto 0);
                Data_out_2: out std_logic_vector(31 downto 0);
                Initi_Done: out std_logic;
                initivalue_done: out std_logic);
end component;

component Second is
```

```
    port(
            clk: in std_logic;
            Adder_1: in std_logic_vector( 31 downto 0);
                Adder_2: in std_logic_vector(31 downto 0);
                Sum_1: out std_logic_vector(31 downto 0)
              );
end component;

component Third is
    port(    clk    : in std_logic;
            reset: in std_logic;
               continue: in std_logic;
            Init_Start: in std_logic_vector(1 downto 0);
            Raddr_3: in std_logic_vector(10 downto 0);
            Waddr_3: in std_logic_vector(10 downto 0);
            Data_in_2: in std_logic_vector(31 downto 0);
          write_en : in std_logic;
            Data_out_3: out std_logic_vector(31 downto 0);
            Initi_Done: out std_logic;
            initivalue_done: out std_logic

                );
end component;

component Fourth is
     port(
            clk: in std_logic;
            Adder_1: in std_logic_vector( 31 downto 0);
                Adder_2: in std_logic_vector(31 downto 0);
                Sum: out std_logic_vector(31 downto 0)
              );
end component;

component SetRaddr
    port( clk : in std_logic;
            reset: in std_logic;
            initvalue_1: in std_logic_vector(10 downto 0);
            initvalue_2: in std_logic_vector(10 downto 0);
            initvalue_3: in std_logic_vector(10 downto 0);
```

```
            length: in std_logic_vector(10 downto 0); -- for example 31, 6, l = 30, k =
25;
            Raddr_1_en: in std_logic;
            Raddr_2_en: in std_logic;
            Raddr_3_en: in std_logic;
            Raddr_1: out std_logic_vector(10 downto 0);
            Raddr_2: out std_logic_vector(10 downto 0);
            Raddr_3: out std_logic_vector(10 downto 0)
                  );

end component;

component SetWrite

        port( clk    : in std_logic;
            Raddr_1: in std_logic_vector(10 downto 0);
            Raddr_3: in std_logic_vector(10 downto 0);
            Waddr_1: out std_logic_vector(10 downto 0);
            Waddr_3: out std_logic_vector(10 downto 0)
           );

end component;


begin
FirstStep: First
port map(
            clk => clk,
              reset => reset,
                  continue => continue,
           Init_Start => Init_Start,
              Raddr_1 => Raddr_1,
              Raddr_2 => Raddr_2,
              Waddr_1      => Waddr_1,
              Data_in_1 => Data_in_1,
          write_en    => write_en_1,
          Data_out_1 => Rdata_1,
              Data_out_2 => Rdata_2,
              Initi_Done => Initi_Done_1,
```

```
            initivalue_done => initivalue_done_1
                    );


SecondStep: Second
      port map(
                clk => clk,
                Adder_1 => Rdata_1,
                      Adder_2 => Rdata_2,
                      Sum_1 => Wdata_3
                   );

ThirdStep: Third
      port map (    clk    => clk,
                 reset => reset,
                        continue => continue,
                  Init_Start => Init_Start,
                    Raddr_3 => Raddr_3,
                    Waddr_3 => Waddr_3,
                    Data_in_2 => Data_in_2,
                  write_en => write_en_3,
                          Data_out_3 => Rdata_3,
                    Initi_Done => Initi_Done_3,
                    initivalue_done => initivalue_done_3
                          );

FourthStep: Fourth
       port map(
                clk => clk,
                Adder_1 => Wdata_3_dly,
                      Adder_2 => Rdata_3,
                      Sum =>Wdata_1
                   );

ReadAdrr: SetRaddr
      port map( clk => clk,
                reset => reset,
            initvalue_1 => initvalue_1,
            initvalue_2 => initvalue_2,
```

```
            initvalue_3 => initvalue_3,
            length => counterlength,
                Raddr_1_en => Raddr_1_en,
                Raddr_2_en => Raddr_2_en,
                Raddr_3_en => Raddr_3_en,
                Raddr_1 => Raddr_1,
                Raddr_2 => Raddr_2,
                Raddr_3 => Raddr_3                    );

WriteAddr: SetWrite

             port map ( clk => clk,
                    Raddr_1 => tempRaddr_1,
                        Raddr_3 => tempRaddr_3,
                        Waddr_1 => Waddr_1,
                      Waddr_3 => Waddr_3
              );
WriteAddrDelay1clk: SetWrite
      port map ( clk => clk,
                  Raddr_1 => Raddr_1,
                  Raddr_3 => Raddr_3,
                  Waddr_1 => tempRaddr_3,
                  Waddr_3 => tempRaddr_1
                  );
Data_in_1 <= Seeds when Init_Start = "00"
                  else Wdata_1;
Data_in_2 <= Seeds when Init_Start = "00"
                  else Wdata_3;
Data <= Wdata_3;
counterlength <= length - 1 when initivalue_Done = '1'
            else "00000000000";
initivalue_done <= initivalue_done_1 and initivalue_done_3;
Initi_Done <= Initi_Done_1 and Initi_Done_3;

process (clk, reset)
begin
      if reset = '0' then
            state<="000";
            Data_valid <= '0';
```

```
        Raddr_1_en    <= '0';
                Raddr_2_en <= '0';
                Raddr_3_en <= '0';
        Write_en_1 <= '0';
        Write_en_3 <= '0';
        Data_valid <= '0';
else
        if clk' event and clk ='1' then

            if continue = '1' then
              case state is
                        when "000" =>
                            if initivalue_Done = '1' then
                                state <="001";
                                initvalue_1 <= (others =>'0');
                                initvalue_2 <= length-k;
                                initvalue_3 <= k;
                            else
                                state <="000";
                            end if;
                        when "001" =>
                            if Initi_Done = '1' then
                                state <="010";
                            else
                                state <="001";
                            end if;
                        when "010" =>
                            Raddr_1_en <= '1';
                            Raddr_2_en <= '1';
                            state <="011";
                        when "011" =>
                            state <= "100";
                        when "100" =>
                            write_en_3 <='1';
                            Data_valid <='1';
                            state <= "101";
                            Raddr_3_en <= '1';
                        when "101" =>
                            state <= "110";
```

```
                    write_en_1<='1';

                when "110" =>
                    state <="110";
                when others =>
                    null;
            end case;
             end if;
            end if;
    end if;
end process;
ack_delay: process(clk,reset)
begin
    if clk'event and clk ='1' then
        Wdata_3_dly <= Wdata_3;
    end if;
end process;
end;
```

Appendix II-- Test Random Numbers Generation Codes

1. Main PALFG TestBench File

```
/*************************************************************
Name of code:    PALFGTestBench.cpp

Purpose of code:    Test Bench File

Author of code:    Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in
the Tennessee Advanced Computing Laboratory.

Copyright (C) 2006 Yu Bi and Gregory D. Peterson

This library is free software; you can redistribute it and/or modify it under the terms of
the GNU Lesser General Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any later version. This library is
distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License along with
this library; if not, write to
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA    02110-1301 USA.

You may also view the GNU Lesser General Public License at
http://www.gnu.org/licenses/lgpl.html

For additional information or queries, send email to gdp@utk.edu.
*************************************************************/
#include "stdafx.h"

#include "SPRNGThread.h"

#include "ReadThread.h"
//#define GENERATORNUMBER 2

int _tmain(int argc, _TCHAR* argv[])
```

```
{

   int i;
   printf( "\n Enter the number of Random Number Generators that you want\n");
   int GeneratorNumber=0;
   int parameter;
   scanf( "%d", &GeneratorNumber);
   printf( "\n Select the type of Random Number Generator as follows\n please type the
NO. from 0 to 10\n");
   printf("\n 0. {1279,861,1,233}\n 1. {17,5,1,10}\n 2. {31,6,1,2} \n 3. {55,24,1,11}\n 4.
{63,31,1,14}\n 5.{127,97,1,21}\n 6.{521,353,1,100}\n 7. {521,168,1,83}\n 8.
{607,334,1,166}\n 9. {607,273,1,105}\n 10. {1279,418,1,208}\n");
   scanf("%d",&parameter);

   CSPRNGThread *asprngtThreads;
   asprngtThreads =    new CSPRNGThread[GeneratorNumber];
   CReadThread rtReadThread(asprngtThreads, GeneratorNumber);

   for (i=0;i<GeneratorNumber;i++)
   {
        asprngtThreads[i].SetID(i);
        asprngtThreads[i].SetNumRNG(GeneratorNumber);
        asprngtThreads[i].SetParam(parameter);
   }
   for (i=0; i<GeneratorNumber; i++)
   { asprngtThreads[i].CreateThread();
   }


   // general seed
      DWORD dwSeed = 0;


   for (i=0; i<GeneratorNumber; i++)
   {
        asprngtThreads[i].SetSeed(dwSeed);
   }

   rtReadThread.CreateThread();
```

```
    Sleep(1000);

    rtReadThread.Terminate();
    WaitForSingleObject(rtReadThread.m_hThread, INFINITE);



    // stop
    for (i=0; i<GeneratorNumber; i++)
    {
      asprngtThreads[i].StopGenerator();
       WaitForSingleObject(asprngtThreads[i].m_hThread, INFINITE);
    }
      return 0;
}
```

2. PALFG Object

```
/*************************************************************
Name of code:    PALFG.cpp

Purpose of code:    PALFG Thread

Author of code:    Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in
the Tennessee Advanced Computing Laboratory.

Copyright (C) 2006 Yu Bi and Gregory D. Peterson

This library is free software; you can redistribute it and/or modify it under the terms of
the GNU Lesser General Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any later version. This library is
distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License along with
this library; if not, write to
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA    02110-1301 USA.

You may also view the GNU Lesser General Public License at
http://www.gnu.org/licenses/lgpl.html

For additional information or queries, send email to gdp@utk.edu.
*************************************************************/
#if !defined(SPRNGTHREAD_H)
   #include "SPRNGThread.h"
#endif
#define GENERATORNUMBER 2
int error;
CSPRNGThread::CSPRNGThread()
   : m_iInitSeed(0), m_bContinue(true),m_RNFIFO(1000)
{
   m_hEventForSeed = CreateEvent(NULL, FALSE, FALSE, NULL);
```

```
    m_bAutoDelete = FALSE;
    m_Param = 0;
    m_bPause = false;
}
CSPRNGThread::~CSPRNGThread()
{
      CloseHandle(m_hEventForSeed);
}
void CSPRNGThread::SetID(int iID)
{
      m_iID = iID;
}
void CSPRNGThread::SetNumRNG(int iNumofRNG)
{
       m_iNumofRNG = iNumofRNG;
}
void CSPRNGThread::SetSeed(int iSeed)
{
      m_iInitSeed = iSeed;
      SetEvent(m_hEventForSeed);
}
void CSPRNGThread::StopGenerator()
{
      m_bContinue = false;
       SetEvent(m_hEventForSeed);
}
 BOOL CSPRNGThread::InitInstance()
{
      CWinThread::InitInstance();
      return TRUE;
}
int
CSPRNGThread::Run()
{
      WaitForSingleObject(m_hEventForSeed, INFINITE);
       if (!m_bContinue)
       return 0;
      m_iRNG = init_rng(m_iID,GENERATORNUMBER,m_iInitSeed,m_Param);
       while (m_bContinue)
```

```
        {
            Random();
            if (m_dwRN_z == 0x4cae3669)
            {
                error = 1;
            };
            WriteRNtoFIFO();
            while(m_RNFIFO.IsFull())
            {
                if (!m_bContinue)
                    break;}
        }
       return 0;
}
void
CSPRNGThread::Random()
{
     m_dwRN_z = get_rn_int(m_iRNG);
}
void CSPRNGThread::WriteRNtoFIFO()
{
     m_RNFIFO.Put(m_dwRN_z);
}
void CSPRNGThread::SetParam(int param)
{
     m_Param=param;
}
```

3. PALFG Execution Function Files

/***

Name of code: PALFGfunc.cpp

Purpose of code: PALFG basic functions

Author of code: Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in the Tennessee Advanced Computing Laboratory.

For additional information or queries, send email to gdp@utk.edu.

***/

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include "LFG.H"
#define INT_MOD_MASK 0xffffffff
#define INT_MASK ((unsigned)INT_MOD_MASK>>1)
#define BITS_IN_INT_GEN 32
```

```
#define MAX_BIT_INT (BITS_IN_INT_GEN-2)
#define INTX2_MASK ((1<<MAX_BIT_INT)-1)
#define GS0 0x372f05ac
#define RUNUP (2*BITS_IN_INT_GEN)


struct vstruct {
        int L;
        int K;
        int LSBS;       /* number of least significant bits that are 1 */
        int first;      /* the first seed whose LSB is 1 */
};

const struct vstruct valid[] = { {1279,861,1,233}, {17,5,1,10}, {31,6,1,2},
{55,24,1,11}, {63,31,1,14}, {127,97,1,21}, {521,353,1,100},
{521,168,1,83}, {607,334,1,166}, {607,273,1,105}, {1279,418,1,208}};



static int bitcnt(int x)

{
   unsigned i=0,y;

   for (y=(unsigned)x; y; y &= (y-1) )
      i++;

   return(i);
}
static void advance_reg(int *reg_fill)

{
/*          the register steps according to the primitive polynomial          */
/*                  (64,4,3,1,0); each call steps register 64 times           */
/*          we use two words to represent the register to allow for integer   */
/*                  size of 32 bits                                           */
```

```
    const int mask = 0x1b;
    int adv_64[4][2];

    int i,new_fill[2];
    unsigned temp;

    adv_64[0][0] = 0xb0000000;
    adv_64[0][1] = 0x1b;
    adv_64[1][0] = 0x60000000;
    adv_64[1][1] = 0x2d;
    adv_64[2][0] = 0xc0000000;
    adv_64[2][1] = 0x5a;
    adv_64[3][0] = 0x80000000;
    adv_64[3][1] = 0xaf;
    new_fill[1] = new_fill[0] = 0;
    temp = mask<<27;

    for (i=27;i>=0;i--)
    {
       new_fill[0] = (new_fill[0]<<1) | (1&bitcnt(reg_fill[0]&temp));
       new_fill[1] = (new_fill[1]<<1) | (1&bitcnt(reg_fill[1]&temp));
       temp >>= 1;
    }

    for (i=28;i<32;i++)
    {
       temp = bitcnt(reg_fill[0]&(mask<<i));
       temp ^= bitcnt(reg_fill[1]&(mask>>(32-i)));
       new_fill[0] |= (1&temp)<<i;
       temp = bitcnt(reg_fill[0]&adv_64[i-28][0]);
       temp ^= bitcnt(reg_fill[1]&adv_64[i-28][1]);
       new_fill[1] |= (1&temp)<<i;
    }

    reg_fill[0] = new_fill[0];
    reg_fill[1] = new_fill[1];
}
```

```
static int get_fill( unsigned *n, unsigned *r, int param, unsigned seed)

{
   int i,j,k,temp[2], length;

   length = valid[param].L;


   temp[1] = temp[0] = n[0]^seed;
   if (!temp[0])
      temp[0] = GS0;


   advance_reg(temp);
   advance_reg(temp);

/*          the first word in the generator is defined by the 31 LSBs of the */
/*                   node number
*/

   r[0] = (INT_MASK&n[0])<<1;
/*          the generator is filled with the lower 31 bits of the shift          */
/*                   register at each time, shifted up to make room for the bits */
/*                   defining the canonical form; the node number is XORed into   */
/*                   the fill to make the generators unique                        */

   for (i=1;i<length-1;i++)
   {
      advance_reg(temp);
      r[i] = (INT_MASK&(temp[0]^n[i]))<<1;
   }
   r[length-1] = 0;
/*          the canonical form for the LSB is instituted here                     */
   k = valid[param].first + valid[param].LSBS;

   for (j=valid[param].first;j<k;j++)
      r[j] |= 1;

   return(0);
```

```
}


static void si_double(unsigned *a,    unsigned *b, int length)

{
   int i;

   a[length-2] = (INTX2_MASK&b[length-2])<<1;
   for (i=length-3;i>=0;i--)
   {
      if (b[i]&(1<<MAX_BIT_INT))
         a[i+1]++;
      a[i] = (INTX2_MASK&b[i])<<1;
   }
}

int get_rn_int(int *genptr)

/*          returns value put into new position                                    */
{
       unsigned new_val,*r0,*r1;
       int hptr,lptr,*hp = &((struct rngen *)genptr)->hptr;
      int lval, kval;
      int test1;

      lval = ((struct rngen *)genptr)->lval;
      kval = ((struct rngen *)genptr)->kval;
       r0 = ((struct rngen *)genptr)->r0;
       r1 = ((struct rngen *)genptr)->r1;
       hptr = *hp;
       lptr = hptr + kval;
             if (lptr>=lval) lptr -= lval;
/*       INT_MOD_MASK causes arithmetic to be modular when integer size is    */
/*                 different from generator modulus
            */
            if (r0[hptr]    == 0x487d988a)
            {
```

```
                test1 = 0;
            }
            r0[hptr] = INT_MOD_MASK&(r0[hptr] + r0[lptr]);
            r1[hptr] = INT_MOD_MASK&(r1[hptr] + r1[lptr]);
            new_val = (r1[hptr]&(~1)) ^ (r0[hptr]>>1);
            if (--hptr < 0) hptr = lval - 1; /* skip an element in the sequence */
            if (--lptr < 0) lptr = lval - 1;
            r0[hptr] = INT_MOD_MASK&(r0[hptr] + r0[lptr]);
            r1[hptr] = INT_MOD_MASK&(r1[hptr] + r1[lptr]);
            *hp = (--hptr < 0) ? lval-1 : hptr;
            return (new_val>>1);
}



static int **initialize(int ngen, int param, unsigned seed, unsigned *nstart, unsigned
initseed)

{
   int i,j,k,l,*order, length;
   struct rngen **q;
   unsigned *nindex;
   int lval;


   length = valid[param].L;
   lval = length;

/*          allocate memory for node number and fill of each generator          */
   order = (int *) malloc(ngen*sizeof(int));
   q = (struct rngen **) malloc(ngen*sizeof(struct rngen *));
   if (q == NULL || order == NULL)
      return NULL;

   for (i=0;i<ngen;i++)
   {
      q[i] = (struct rngen *) malloc(sizeof(struct rngen));
      if (q[i] == NULL)
         return NULL;
```

```
//    q[i]->rng_type = rng_type;
      q[i]->hptr = length - 1;
      q[i]->si = (unsigned *) malloc((length-1)*sizeof(unsigned));
      q[i]->r0 = (unsigned *) malloc(length*sizeof(unsigned));
      q[i]->r1 = (unsigned *) malloc(length*sizeof(unsigned));
      q[i]->lval = length;
      q[i]->kval = valid[param].K;
      q[i]->param = param;
      q[i]->seed = seed;
      q[i]->init_seed = initseed;
      //q[i]->gentype = GENTYPE;

      if (q[i]->r1 == NULL || q[i]->r0 == NULL || q[i]->si == NULL)
         return NULL;
   }
/*          specify register fills and node number arrays                    */
/*          do fills in tree fashion so that all fills branch from index     */
/*                   contained in nstart array                               */
   q[0]->stream_number = nstart[0];
   si_double(q[0]->si,nstart,length);
   get_fill(q[0]->si,q[0]->r0,param,seed);
   q[0]->si[0]++;
   get_fill(q[0]->si,q[0]->r1,param,seed);

   i = 1;
   order[0] = 0;
   if (ngen>1)
      while (1)
      {
         l = i;
         for (k=0;k<l;k++)
         {
           nindex = q[order[k]]->si;
           q[i]->stream_number = nindex[0];
           si_double(nindex,nindex, length);
           for (j=0;j<length-1;j++)
                 q[i]->si[j] = nindex[j];
           get_fill(q[i]->si,q[i]->r0,param,seed);
```

```
            q[i]->si[0]++;
            get_fill(q[i]->si,q[i]->r1,param,seed);
            if (ngen == ++i)
                  break;
          }

          if (ngen == i)
            break;

          for (k=l-1;k>0;k--)
          {
            order[2*k+1] = l+k;
            order[2*k] = order[k];
          }
          order[1] = l;
        }

free(order);

for (i=ngen-1;i>=0;i--)
{
    k = 0;
    for (j=1;j<lval-1;j++)
          if (q[i]->si[j])
                k = 1;
          if (!k)
                break;
    for (j=0;j<length*RUNUP;j++)
        get_rn_int((int *)(q[i]));
}

while (i>=0)
{
    for (j=0;j<4*length;j++)
        get_rn_int((int *)(q[i]));
    i--;
}

return((int **)q);
```

```
}

/*************************************************************************
***/
/*************************************************************************
***/
/*                    INIT_RNG's: user interface to start things off                    */
/*************************************************************************
***/
/*************************************************************************
***/

int *init_rng(int gennum,    int total_gen,    int seed, int param)

{
   int doexit=0,i,k, length;
   int **p=NULL, *rng;
   unsigned *nstart=NULL,*si;
   int lval, kval;
   int gseed = 0;


/*          gives back one generator (node gennum) with updated spawning          */
/*          info; should be called total_gen times, with different value          */
/*          of gennum in [0,total_gen) each call                                  */


   seed &= 0x7fffffff;       /* Only 31 LSB of seed considered */


/*          check whether generators have previously been defined                 */
/*          guard against access while defining generator parameters for          */
/*                    the 1st time
*/
   length = valid[param].L; /* determine parameters      */
   k = valid[param].K;


      lval = length; /* determine parameters      */
```

```
        kval = k;
        gseed = seed^GS0;

/*          define the starting vector for the initial node                 */
    nstart = (unsigned *) malloc((length-1)*sizeof(unsigned));
    if (nstart == NULL)
        return NULL;

    nstart[0] = gennum;
    for (i=1;i<length-1;i++)
        nstart[i] = 0;
    p = initialize(1,param,seed^GS0,nstart,seed);    /* create a generator    */
    if (p==NULL)
        return NULL;
    ((struct rngen *)(p[0]))->stream_number = gennum;
/*          update si array to allow for future spawning of generators      */
    si = ((struct rngen *)(p[0]))->si;
    while (si[0] < total_gen && !si[1])
        si_double(si,si,length);
    free(nstart);
    rng = p[0];
    free(p);
    return rng;
}
```

B. Seed Initialization Algorithm Codes

```
/************************************************************
Name of code:    initialize.cpp

Purpose of code:    Seeding Alogorithm

Author of code:    Yu Bi

Developed under the guidance of Gregory D. Peterson at The University of Tennessee in
the Tennessee Advanced Computing Laboratory.

Copyright (C) 2006 Yu Bi and Gregory D. Peterson

This library is free software; you can redistribute it and/or modify it under the terms of
the GNU Lesser General Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any later version. This library is
distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License along with
this library; if not, write to
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA    02110-1301 USA.

You may also view the GNU Lesser General Public License at
http://www.gnu.org/licenses/lgpl.html

For additional information or queries, send email to gdp@utk.edu.
************************************************************/

#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
   int i;
   printf( "\n Enter the number of Random Number Generators that you want\n");
   int GeneratorNumber=0;
```

```
    int parameter;
    char sz0[16];
    char sz1[16];
    FILE * stream0;
    FILE * stream1;
    FILE * seed0;
    FILE * seed1;
    FILE * seed2;
    FILE * seed3;
    FILE * seed4;
    FILE * seed00;
    FILE * seed11;
    unsigned num;
    unsigned k;
    unsigned* rn0;
    unsigned* rn1;
    _int16 temp0;
    _int16 temp1;
    int odd;
    int kodd;
    unsigned temp;
    printf( "\n Enter the Type NO \n");
    printf( "\n Select the type of Random Number Generator as follows\n please type the
    NO. from 0 to 10\n");
    printf("\n 11. {1279,861,1,233}\n 1. {17,5,1,10}\n 2. {31,6,1,2} \n 3. {55,24,1,11}\n 4.
    {63,31,1,14}\n 5.{127,97,1,21}\n 6.{521,353,1,100}\n 7. {521,168,1,83}\n 8.
    {607,334,1,166}\n 9. {607,273,1,105}\n 10. {1279,418,1,208}\n");
    scanf("%d",&parameter);
    sprintf(sz0, "seed0%d.txt", parameter);
    sprintf(sz1, "seed1%d.txt", parameter);
/*-----------------------------------------------------*/
    if( (stream0 = fopen( sz0, "r+t" )) != NULL )
      {
          /* Attempt to read in 25 characters */
          fscanf( stream0, "%u",&num);
         fscanf( stream0, "%u",&k);
         rn0 = new unsigned [num];
          for ( i=0; i<num; i++)
         {
```

```
            fscanf( stream0, "%x",&rn0[i]);
        }
        fclose( stream0 );
    }
    else
        printf( "File could not be opened\n" );
/*-------------------------------------------------------------*/

    if( (stream1 = fopen( sz1, "r+t" )) != NULL )
    {
        /* Attempt to read in 25 characters */
        fscanf( stream1, "%u",&num);
       fscanf( stream1, "%u",&k);
       rn1 = new unsigned [num];
        for ( i=0; i<num; i++)
        {
            fscanf( stream1, "%x",&rn1[i]);
        }

        fclose( stream1 );
    }
    else
        printf( "File could not be opened\n" );

/*----------------------Initialize the RNG X----------------------*/
    int j=0;
    kodd = k %2;
    if (kodd)
    {
      // K= Odd, follow the initilization rule of the l=odd, k=odd;
          odd = i %2;
          for ( i=0; i< k; i++)
          {
              odd = i %2;
             if (odd)
             {
                  j = num-k+i;
                if (j>num-1)
                      j = i-k;
```

```
                    rn0[i]= rn0[i]+rn0[j];
                }
            }
        }
        else
        {
// K= Even, follow the initilization rule of the l=odd, k=even;

        for (i= 0; i< num-k-1; i++)
        {
          odd = i %2;
          if (odd)
          {
                j = num-k+i;
                if (j>num-1)
                    j = i-k;
                rn0[i]= rn0[i]+rn0[j];
          }
        }
        }
/*-------------------Intialize the RNG Y-----------------------------------*/
        j=0;
        kodd = k %2;
        if (kodd)
        {
                odd = i %2;
                for ( i=0; i< k; i++)
                {
                    odd = i %2;
                  if (odd)
                  {
                        j = num-k+i;
                      if (j>num-1)
                            j = i-k;
                      rn1[i]= rn1[i]+rn1[j];
                  }
                }
        }
        else
```

```
        {
       for (i= 0; i< num-k-1; i++)
       {
         odd = i %2;
         if (odd)
         {
               j = num-k+i;
               if (j>num-1)
                     j = i-k;
               rn1[i]= rn1[i]+rn1[j];
         }
       }
        }
/*-----------------------Save the processed seeds into txt files---------*/
        seed0 = fopen( "seed0.txt", "w+t" );
        seed1 = fopen( "seed1.txt", "w+t" );
        seed00 = fopen( "seed00.txt", "w+t" );
        seed11 = fopen( "seed11.txt", "w+t" );
        seed2 = fopen( "seed2.txt", "w+t" );
        seed3 = fopen( "seed3.txt", "w+t" );
        seed4 = fopen( "crayseeds.txt", "w+t" );
        fprintf(seed0, "%u\r\n", num);
        fprintf(seed0, "%u\r\n", k);
        fprintf(seed1, "%u\r\n", num);
        fprintf(seed1, "%u\r\n", k);
        fprintf(seed00, "%u\r\n", num);
        fprintf(seed00, "%u\r\n", k);
        fprintf(seed11, "%u\r\n", num);
        fprintf(seed11, "%u\r\n", k);
        for (i=0; i<num; i++)
        {
              temp = rn0[i]& 0xff000000;
              temp0 = temp >> 24;
              fprintf(seed0, "%u\r\n", temp0);
              temp = rn0[i]&0x00ff0000;
              temp0 = temp >> 16;
              fprintf(seed0, "%u\r\n", temp0);
              temp = rn0[i]&0x0000ff00;
              temp0 = temp >> 8;
```

```
            fprintf(seed0, "%u\r\n", temp0);
            temp0 = rn0[i]&0x000000ff;
            fprintf(seed0, "%u\r\n", temp0);
            temp = rn1[i]& 0xff000000;
            temp1 = temp >> 24;
            fprintf(seed1, "%u\r\n", temp1);
            temp = rn1[i]&0x00ff0000;
            temp1 = temp >> 16;
            fprintf(seed1, "%u\r\n", temp1);
            temp = rn1[i]&0x0000ff00;
            temp1 = temp >> 8;
            fprintf(seed1, "%u\r\n", temp1);
            temp1 = rn1[i]&0x000000ff;
            fprintf(seed1, "%u\r\n", temp1);
            fprintf(seed00, "%x\r\n", rn0[i]);
            fprintf(seed11, "%x\r\n", rn1[i]);
            fprintf(seed2, "0x%x, ", rn0[i]);
            fprintf(seed3, "0x%x, ", rn1[i]);
            fprintf(seed4, "0x%.8x%.8x,", rn1[i], rn0[i]);
        }
/*-------------------------------------------------------------*/
        fclose(seed0);
        fclose(seed1);
         delete    [] rn0;
        delete [] rn1;
        return 0;
}
```

Vita

Yu Bi was born in Shanghai, China. She started to study Electronics Engineering at East China Normal University in 1997. She earned the top University Fellowship and Huawei Fellowship during her undergraduate study. In 2001, she received her Bachelor of Science degree and is currently pursuing her Master of Science degree in Computer Engineering at UTK.