8-2007

Hardware Accelerated Scalable Parallel Random Number Generation

Junkyu Lee
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Junkyu Lee entitled "Hardware Accelerated Scalable Parallel Random 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 Electrical Engineering.

Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Itamar Elhanany, Robert J. Harrison

Accepted for the Council:
Dixie L. Thompson

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 JunKyu Lee entitled “Hardware Accelerated Scalable Parallel Random 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 Electrical Engineering.

Gregory D. Peterson
Major Professor

We have read this thesis
and recommend its acceptance:

Itamar Elhanany

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.)

Hardware Accelerated Scalable Parallel Random Number Generation

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

JunKyu Lee
August 2007

Acknowledgements

First of all I would like to thank Dr. Gregory D. Peterson for his constant support during this work and his hearty advice. Second, I would like to thank Dr. Robert J. Harrison for giving me valuable suggestions for this work. I would like to thank Dr. Itamar Elhanany for serving on my committee. I would like to thank Dr. Don Bouldin for teaching me FPGA design through his courses. Finally, I would like to thank my research members, Akila Gothandaraman, Junquing Sun, Depeng Yang, and Saumil Merchant for their help and encouragement.

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

This thesis is dedicated to my parents.

Abstract

The Scalable Parallel Random Number Generators library (SPRNG) is widely used due to its speed, quality, and scalability. Monte Carlo (MC) simulations often employ SPRNG to generate large quantities of random numbers. Thanks to fast Field-Programmable Gate Array (FPGA) technology development, this thesis presents Hardware Accelerated SPRNG (HASPRNG) for the Virtex-II Pro XC2VP30 FPGAs. HASPRNG includes the full set of SPRNG generators and provides programming interfaces which hide detailed internal behavior from users. HASPRNG produces identical results with SPRNG, and it is verified with over 1 million consecutive random numbers for each type of generator. The programming interface allows a developer to use HASPRNG the same way as SPRNG. HASPRNG introduces 4-70 times faster execution than the original SPRNG. This thesis describes the implementation of HASPRNG, the verification platform, the programming interface, and its performance.

Table of Contents

List of Tables

List of Figures

1. Introduction

Random numbers are required in a wide variety of applications such as data encryption, circuit testing, system simulation, and Monte Carlo (MC) simulations [11]. MC simulations are widely used for solving mathematical problems by using random processes. Random number generators play a critical role in MC simulations. The Scalable Parallel Random Number Generators library (SPRNG) was designed to support parallel MC applications on scalable and distributed architectures [1]. SPRNG includes several random number generators which are qualitatively distinct, well-tested, and scalable. These random number generators provide random number streams to parallel processes. One of the most important considerations for simulation applications is random number generation speed. Hardware implementation of the generators is able to improve their speed.

This thesis presents a set of Hardware Accelerated Scalable Parallel Random Number Generators (HASPRNG) implemented in FPGAs. This thesis starts with related work related to the hardware implementation of random number generators in Chapter 2. It then describes the HASPRNG implementation methodology in detail including VHDL descriptions in Chapter 3. Also, it will inform a reader of how to implement a verification platform and the programming interface in Chapter 4 and 5. Chapter 6 describes the HASPRNG performance results. Finally Chapter 7 describes conclusions and future work.

1.1. Random number generators

Random numbers are used for applications in many areas such as simulation, game-playing, cryptography, statistical sampling, evaluation of multiple integrals, and computations in statistical physics [11, 18]. Since each application has different criteria of good randomness, a variety of random number generators have been developed.

Three types of number sequences are introduced for random number representations. One of them is the random-number sequence generated by a truly random process. A more practical sequence is the pseudo-random sequence generated by a deterministic process that is intended to imitate a random sequence. The last one is the quasi-random sequence that is not random but has useful properties similar to randomness in certain applications such as the quasi-Monte Carlo method [21]. Random number generators are characterized by correlation properties. Random number generators must be tested for their correlations in order to apply them to real applications [16].

1.2. Why SPRNG?

MC simulations consume huge quantities of random numbers [17, 22]. Many random number generators currently used are defective in large-scale MC simulation applications [1]. The SPRNG has passed rigorous tests to demonstrate the good statistical properties of its random number generators [1, 2]. A high quality random number generator produces numbers according to a distribution that is close to a uniform distribution. Various tests check a random number stream for the uniformity of the stream and for correlations between numbers in the stream.

Parallel random number generator tests of SPRNG include the additional restriction that there should not be correlations between different random number streams. In order to test correlation properties, the tests consider modifications of serial streams. The modifications of serial streams are described in Figure 1 [2].

If we interleave two streams to make a new stream, this new stream should pass the tests if the original streams (X, Y, Z) were not correlated. The results have passed tests based on Kolmogorov-Smirnov and Chi-square tests which are used for check for a uniform distribution.

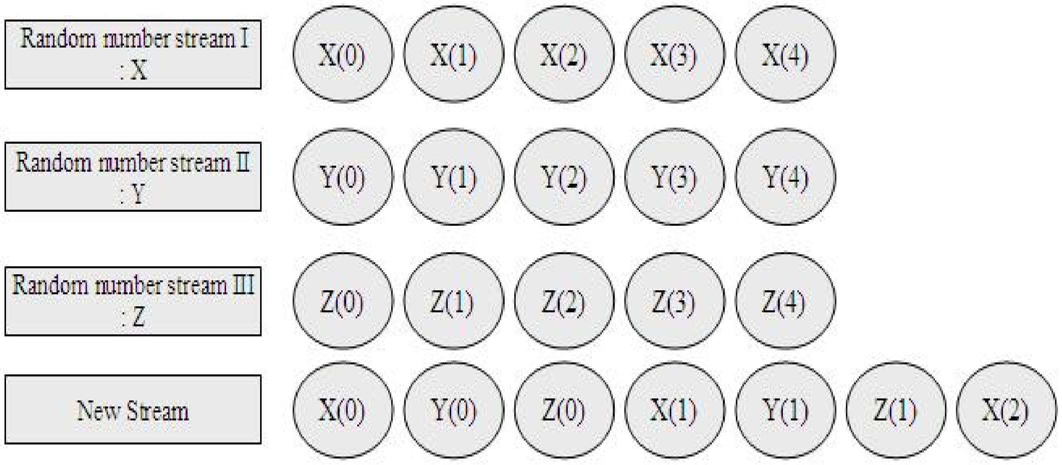


Figure 1. Interleaving Streams [2]

Besides the fact that SPRNG has passed rigorous tests, NCSA (National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign) has declared SPRNG the best random number generators [6]. Now, SPRNG is widely used by many research laboratories and universities around the world [2].

1.3. High Performance Reconfigurable Computing

High performance reconfigurable computers (HPRCs) are parallel computing systems containing multiple processors, and multiple FPGAs [23, 8]. HPRC offers high performance users with the potential of increased performance and flexibility for a wide range of computationally demanding problems [25]. A HPRC platform consists of a number of computing nodes connected by interconnection networks (ICNs). Figure 2 shows the HPRC architecture [24]. The ICN can be a switch, hypercube, mesh, systolic array or other architecture. A reconfigurable computing (RC) element such as FPGAs is connected to computer nodes or other RC elements by ICNs.

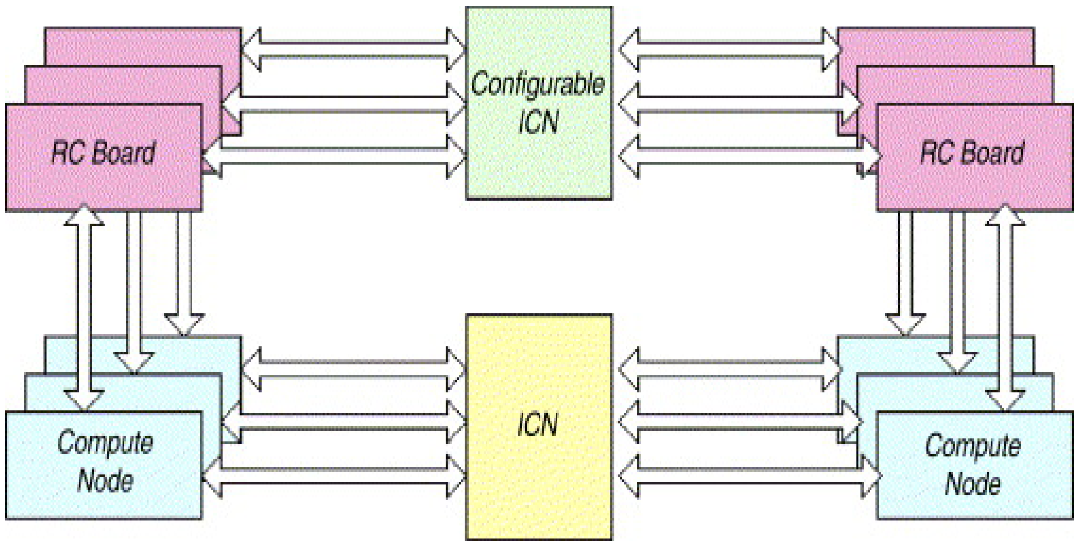


Figure 2. HPRC Architecture [24]

Application designs in HPRCs employ the FPGAs to execute the portion of the application that takes most of the execution time [23]. For example, microprocessors perform the operations that are not efficient for reconfigurable logic such as loops, branches, and possible memory accesses, while computational cores are performed in the reconfigurable hardware in order to improve performance [25]. HPRCs have the potential to take advantage of coarse-grained functional parallelism and fine-grained instruction-level parallelism through optimized execution on FPGAs [23]. Reconfigurable hardware implementations have shown significant speedup over software-only solutions [24]. Besides speedup, HPRC introduces a reduction in size, and improves flexibility and upgradeability [15]. It also reduces energy and power consumption, since the circuitry in HPRC is optimized for the applications [15, 23].

1.4. Motivation

MC simulation techniques require fast and reliable random number sources [9]. Many of the popular pseudo random number generators have been found to be defective for real applications [1]. Random number generators requirements include good randomness properties, portability, reproducibility, performance speed, and a very long period [9]. SPRNG is fit for MC simulations, and has been tested under rigorous conditions [2].

In the past, most random number generators were done by software implementations [11]. According to Moore's law, the hardware is getting denser and faster. It is desirable to improve the generation speed by implementing the hardware random number generators for simulation applications, since generating random numbers takes a considerable amount of the execution time for applications which require huge quantities of random numbers. FPGAs have several advantages in terms of speedup, energy, power, and flexibility.

Based on the good quality of SPRNG and advantages from FPGAs, HASPRNG was developed along with a convenient programming interface.

Chapter 2. Related Work

This chapter describes previous research into effective FPGA implementations of random number generators.

2.1. Random number generator design in FPGA for simulation applications

Besides SPRNG, numerous random number generators have been investigated. One project explored the design methods for hardware acceleration of pseudo-random number generation for simulation applications [10]. In order to improve the performance of simulation applications using substantial quantities of random numbers, it is effective to generate the random numbers in FPGA hardware while the simulator is executing. When a random number is required for the simulation application, the simulator reads a number from a buffer containing the random numbers previously generated by the hardware. An optimized hardware and software interface design will be required for high performance. The research recommends a designer design a hardware-accelerated random number generator using the fixed point representation and then convert the random numbers to floating point by using a lookup table and a linear interpolation scheme. They estimated speedup for three different simulation applications when employing this method. The estimated speedup is around 4 times for a Pi-estimator application, 5 times for a MC integrator application, and 1.02 times for a stochastic simulation application. The estimated speedup does not consider hardware overhead such as the time it takes for generating random number numbers in the FPGA and reading data from the FPGA memory. The authors note hardware overhead depends on the FPGA and the method of interfacing to the main system. The research also illustrates the trade off between hardware size and the speed and accuracy of the random numbers [10].

2.2. Random number generator design in FPGA for different data size

Another paper also introduces design techniques for FPGA based random number generators [11]. The paper provides some methods that are appropriate for FPGA implementations, and discusses their relative benefits. The paper describes design techniques to implement random number generators according to a random number data size. They address two techniques to implement one bit random number generator. One of the techniques is to employ thermal noise, since the thermal noise of electronic components exhibits randomness. The noise is converted to a voltage signal, then compared to a mean value of the voltage values, and encoded to a one-bit number. Another technique for implementing one bit random number generator is to employ an n-bit linear feedback shift register (LFSR), and then take a single bit out of n bits, but this method does not provide good randomness. The paper also addresses three methods for implementing multiple-bit random number generators, which are expressed by the accumulation method, leap-forward LFSR, and lagged Fibonacci methods. The accumulation method is an extension of previous one-bit implementation methods. The accumulation method can obtain an n-bit random number by accumulating the one-bit value n times. This method can be realized by employing several copies of the one bit random number generator hardware, or by repeating the one bit generator several times. In the lagged Fibonacci method, random numbers are generated by the following equation:

$$X(n) = X(n-k) \text{ Op. } X(n-l)$$

In this equation, X(n) is a current random number, and *l* and *k* are time lags. Op. is an operator which can be bit-wise XOR, addition, or multiplication. The equation shows that *l* previous values have to be kept. The *l* previous values are stored in memory or a set of registers. The two random number generators (Modified lagged Fibonacci generator and Multiplicative lagged

Fibonacci generator) in HASPRNG are implemented by using this method. The dual port RAM is employed to store the *l* previous values.

2.3. FPGA based random number generators

The Lehmer random number generator has been implemented by hardware as well [12]. The property of the Lehmer random number generator is characterized by the following equation:

$$X(n+1) = a \times X(n) \bmod M$$

In this equation, the modulus *M* is 2^{31}-1, and the multiplier *a* is 7^5 (= 16807). A random number is represented by a 31 bit number. The research for the Lehmer random number generator implementation shows how to transform the above equation into another form for hardware implementation. They unfold the n-bit data into a binary notation, and transform the multiplication operation into add and subtract operations. The Lehmer random number generator employs three carry-save adders, one carry-propagate subtracter, and one carry-propagate adder [12]. This generator's multiplier is fixed. The generators period at most is the modulus-1, 2^{31}-2 when the modulus number is a prime number [1, 2]. The characteristic is very similar to prime modulus linear congruential generator (PMLCG) in SPRNG. PMLCG uses 2^{61}-1 as its modulus number and its multiplier is chosen to maximize the period in SPRNG which means the PMLCG has longer period than the Lehmer generator.

The Linear Feedback Shift Register generator (LFSR) employs a shift register whose input bit comes from a linear function of its previous state [31]. The initial state is decided by the initial values in the register. A random number generated by the generator is deterministic. The Linear Feedback Shift Register generator (LFSR) has the period related to its data size. If the data size is m, the maximum period is 2^m -1 [1, 31]. The main issue of the LFSR comes from linear dependencies [30]. The LFSR is inappropriate for scientific computing applications which employ parallel generation [3].

The True Random Number Generator (TRNG) relies on physical processes such as thermal noise to produce a random number. So, the random number is unpredictable [28]. FPGA-based TRNGs are available. Even though this generator is excellent for cryptographic purposes, it is not generally useful for simulation applications because it is not reproducible without storing the previous data which makes the generation speed low. If a simulation application consumes huge quantities of random numbers, the generator need substantial amount of memories in order to save the random numbers.

The universal random number generator (URNG) can generate random numbers with uniform, exponential, and Gaussian distribution [29]. It is suited for a communication, and signal simulation applications. The generator is implemented in Xilinx FPGA XC2VP500. The experimental results show the good performance and good statistical results. It is interesting that the generator can provide several distributions, but the paper did not provide parallel generation properties which will be useful in parallel MC simulation applications. Also, the paper did not mention the specific period for the generator.

The modified lagged Fibonacci generator (Modified LFG) in SPRNG is characterized by XOR bit operations of the two additive LFG (ALFG) products. The modified LFG can be expressed by the following equation [1, 2, 3, 20]:

$$Z(n) = X'(n) \text{ XOR } Y'(n)$$

$$X(n) = X(n-k) + X(n-l) \text{ (Mod } 2^{32})$$

$$Y(n) = Y(n-k) + Y(n-l) \text{ (Mod } 2^{32})$$

where l and k are called the lags of the generator, and the generator follows the convention of $l > k$. X(n) and Y(n) are 32 bit random numbers from the two ALFG. X'(n) is obtained from X(n) by setting the least significant bit to 0. Y'(n) is obtained from Y(n) by shifting right by one bit [2, 3, 4]. The period of the modified LFG is $2^{31}(2^{l}-1)$ where l is the lag. Figure 3 describes the generator architecture [4]. The two implementations are employed for the generator according to a lag factor.

2.4. Summary

Several FPGA based random number generators have been released. However, most of them are not appropriate for the MC simulation applications, since they are not verified on parallel generation test. Parallel generation test is one of the most important tests for science computing applications employing HPRC. For the MC simulation applications, the FPGA based random number generators should provide good randomness properties and reproducibility on the parallel generation. Even though TRNG has good randomness properties, but lacks reproducibility.

SPRNG has been verified on the parallel generation test. The good properties of the parallel generation in HASPRNG are inherited by SPRNG. Also HASPRNG has reproducibility because it consists of pseudo random number generators. The HASPRNG can be one of the best random number generators for FPGA based MC simulation applications.

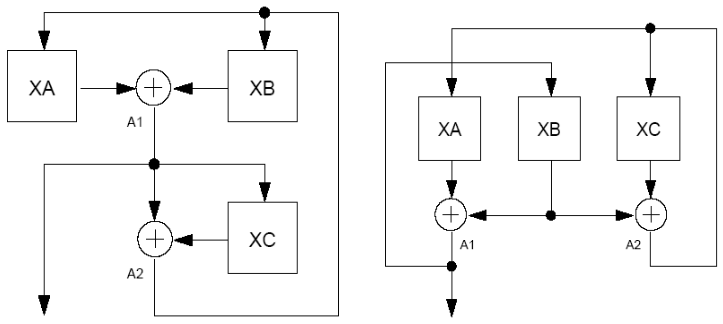


Figure 3. Modified LFG Architecture (Odd-Odd / Odd-Even) [4]

3. HASPRNG Implementations

The HASPRNG development is discussed in this chapter. This chapter describes SPRNG characteristics, the design software and FPGA hardware resources, the implementations approach methods, the implementation architectures, and VHDL descriptions.

3.1. Characteristics of Scalable Parallel Random Number Generators

SPRNG provides two types of random numbers. The first type is integer random numbers, which are represented by 31 bits. The second type is double precision random numbers, which are represented in the range [0,1) [2]. The SPRNG source codes show that the two types of random numbers come from a random number. The difference between the two is the way to express the random number. For example, source codes for the 48 bit linear congruential generator shows that a double precision random number is obtained by dividing the random number by 2^{48}. HASPRNG is designed for integer type random numbers, and it returns a 31 bit random number.

3.1.1. 48 bit Linear Congruential Generator with Prime Addend

A linear congruential generator is commonly used for pseudo random number generation. The linear congruential generator was first proposed for use by Lehmer in 1949 [1]. The characteristic equation of a 48 bit Linear Congruential Generator with prime addend (48 bit LCG) is described by the following equations:

$$Z(n) = a \times Z(n-1) + p \ (\text{Mod } 2^{48})$$

where Z(n) is the nth random number, *p* is a prime number, and *a* is a multiplier coefficient. The prime number is chosen by a function in SPRNG [1, 2, 5]. The modulus *m* can determine the period of an LCG. The period of the generator is 2^{48}.

3.1.2. 64 bit Linear Congruential Generator with Prime Addend

The 64 bit Linear Congruential Generator with prime addend (64 bit LCG) has similar characteristics to the 48 bit LCG. The generator chooses 2^{64} for the modulus value. The multiplier *a* and prime number *p* for this generator are different from those for the 48 bit LCG. The generator is characterized by the following equation:

$$Z(n) = a \times Z(n-1) + p \ (\text{Mod } 2^{64})$$

where Z(n) is the n^{th} random number, *p* is a prime number, and *a* is the multiplier coefficient. The prime number is chosen by a function from the SPRNG. The period of the generator is 2^{64}.

3.1.3. Combined Multiple Recursive Generator

The Combined Multiple Recursive Generator (CMRG) is characterized by the following equations:

$$Z(n) = X(n) + Y(n) \times 2^{32} \ (\text{Mod } 2^{64})$$

$$Y(n) = 107374182 \times Y(n-1) + 104480 \times Y(n-5) \ (\text{Mod } 2^{31}-1)$$

where X(n) is generated by the 64 bit LCG and Z(n) is the resulting random number [2]. The period of the generator is 2^{219}.

3.1.4. Multiplicative Lagged Fibonacci Generator

The multiplicative lagged Fibonacci generator (MLFG) satisfies the following equation:

$$Z(n) = Z(n-k) \times Z(n-l) \ (\text{Mod } 2^{64})$$

where *k* and *l* are time lags and Z(n) is the resulting random number [1, 2, 5, 18]. The period of the generator is $2^{61} \times (2^{l} - 1)$ where l is the lag.

3.1.5. Prime Modulus Linear Congruential Generator

The prime modulus linear congruential generator (PMLCG) is represented by the following equation:

$$Z(n) = a \times Z(n-1) \times 2^{32} \ (\text{Mod } 2^{61}-1)$$

where the *a* is a multiplier parameter and Z(n) is the resulting random number [2]. The period is 2^{61}-2 (Modulus number - 1) [1]. The multiplier coefficient is chosen from SPRNG to get the maximum period.

3.2. Implementations

All the implementations of HASPRNG consider portability, seamless data transfer, and programming interface. HASPRNG implementations employ software interface registers so that they can obtain all the initial parameters inputs. Obtaining all the input parameters through the software interface registers makes it easy to port HASPRNG to other platforms since input parameters do not affect internal behavior. The implementations also use one enable signal and one control signal for all the generators. The enable and control signals play a role to activate and stop the generator operation without changing internal values to make it possible to transfer data seamlessly. The seamless data transfer is able to provide a programming interface in which a user interfaces HASPRNG with the same way as SPRNG. The generators have only two outputs: a random number and the random number valid signal. Those two output data store a random number to a block memory.

3.2.1. Implementation Resources

This section describes the software design tools used for this work such as VHDL design, synthesis, and place and route. Hardware resources also are described in this section.

3.2.1.1. VHDL Designs

ModelSim is used for debugging the VHDL implementations of HASPRNG. ModelSim is able to compile the VHDL source codes and show the results with a waveform window. ModelSim is compatible with Xilinx hardware libraries, such as COREGEN modules. The signal transactions were checked on the waveform window in the ModelSim.

3.2.1.2. Synthesis

Once a design is verified through the ModelSim waveform window, the VHDL source code is synthesized. In the synthesis process, RTL level descriptions are converted to gate level descriptions. The first three generators (48 bit/64 bit LCG, and MLFG) are synthesized by Xilinx Synthesis Tool (XST). The other two generators (CMRG, and PMLCG) are synthesized by the Synplicity tool to improve allowable clock frequency using synthesis options.

3.2.1.3. Place and Route

After the synthesis process, the modules go through the Place And Route (PAR) process. The Xilinx PAR tool is employed for all the generator implementations. The PAR tool reports the maximum allowable frequency for each generator.

3.2.1.4. Xilinx University Program (XUP) - XC2VP30 Board

The XUP is designed for engineering education. The XUP features are described as follows [14]:

1. Virtex-II Pro XC2VP30 FPGA with 30,816 Logic Cells, 136 18-bit multipliers, 2,448Kb of block RAM, and two PowerPC Processors
2. DDR SDRAM DIMM that can accept up to 2Gbytes of RAM
3. 10/100 Ethernet port
4. USB port
5. Compact Flash card slot
6. XSGA Video port
7. Audio Codec
8. SATA, and PS/2, RS-232 ports
9. High and Low Speed expansion connectors with a large collection of available expansion boards

One of the two PowerPC processors is used to give input signals to the FPGA, and read data from the FPGA. The SDRAM is used to load an executable file. The USB port is used to have a communication between the host PC and the Power PC. For example, the Xilinx Microprocessor Debug (XMD) tool is used to load the executable file to the SDRAM, and execute the file through the USB port [13]. The RS-232 port is used to check the results through a HyperTerminal.

3.2.2. Implementation Approach

The approach for the implementation is divided into two parts in general. The first part is to get the correct algorithms from SPRNG before hardware implementation for the generators. The second part is to implement HASPRNG using VHDL for simulation, synthesis, and PAR. The approach for the implementations is described as Figure 4.

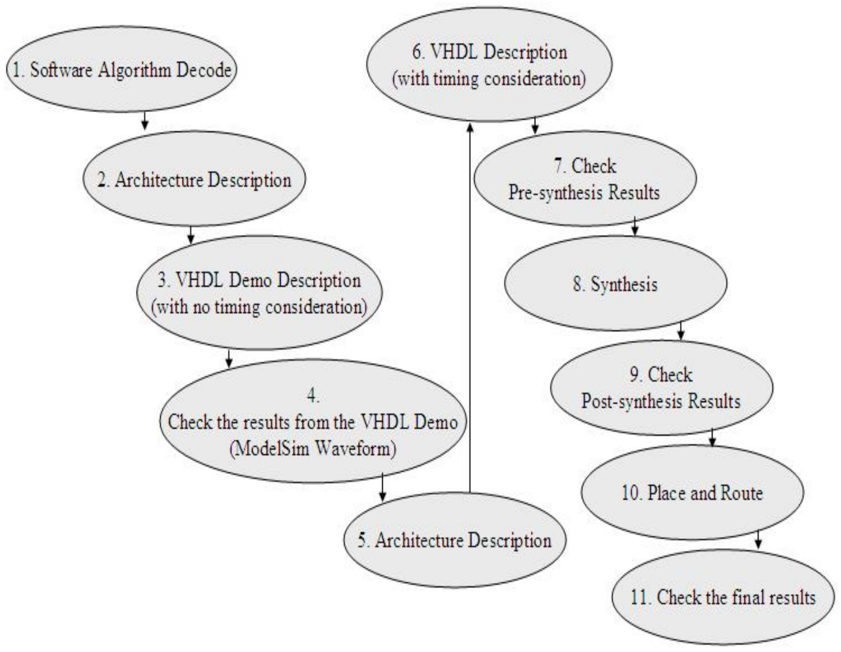

1. Software Algorithm Decode
2. Architecture Description
3. VHDL Demo Description (with no timing consideration)
4. Check the results from the VHDL Demo (ModelSim Waveform)
5. Architecture Description
6. VHDL Description (with timing consideration)
7. Check Pre-synthesis Results
8. Synthesis
9. Check Post-synthesis Results
10. Place and Route
11. Check the final results

Figure 4. Implementation Approach

The left half of Figure 4 represents the steps to get the correct algorithms from the SPRNG for the implementations, and the right half represents the hardware implementation process. The approach minimizes the time it takes to implement the algorithms for the random number generation. The random number generators in SPRNG do not have complicated algorithms. Once the algorithms for a generator are decoded, they are described as a hardware architecture to make a VHDL description for a generator. The hardware architecture does not include timing consideration. So, it does not take long to transform the architecture into a VHDL description. Once the VHDL results are verified with the ModelSim waveform, the architecture is revised to make a VHDL description including timing. When the VHDL description is complete, the results are checked through the ModelSim waveform again. When the results are verified, the VHDL description is synthesized and a bit stream is generated. The final results are checked by a HyperTerminal window.

3.2.3. SPRNG Generator Algorithms

SPRNG provides the source codes for its generators. In order to produce identical results as SPRNG, HASPRNG must obtain initial coefficients values. The function we need to explore first for the implementation is the integer random number generation function in the SPRNG. The core parts of the source code will be described in the following sections. All the random number generators return 31 bit data while concatenating '0' to the most significant bit. Hence, all the random numbers they generate are positive 32bit integers.

3.2.3.1. 48 bit Linear Congruential Generator with Prime Addend

The integer random number generation for the 48 bit LCG is characterized by three functions as follows [2]:

```
/*** Source Code from SPRNG [2] ***/
int LCG::get_rn_int()
   {multiply();  return ((unsigned LONG64) seed) >> 17;}
void LCG::multiply()
   { mult_48_64(&seed,&multiplier,&seed);  seed +=  prime;  seed &= LSB48;}
void mult_48_64(unsigned long int * a, unsigned long int * b, unsigned long int *c)
   { *c = (*a * *b);}
```

The integer random number generation function (get_rn_int()) calls another function (multiply()), which calls another function (mult_48_64()). The last function gets three pointer arguments. The three arguments represent three addresses for a seed, a multiplier coefficient, and an output. The last function multiplies a seed by a multiplier coefficient, and produces the output. The output is modified by adding the prime, and then is masked by 48 bits in the second function. Finally, the generation function returns 31 bits of the modified results.

3.2.3.2. 64 bit Linear Congruential Generator with Prime Addend

The following two functions are sufficient to characterize the 64 bit LCG implementation [2]:

```
/*** Source Code from SPRNG [2] ***/
int LCG64::get_rn_int()
   { advance_state();    return state>>33; }
void LCG64::advance_state()
   {state = state*multiplier + prime;}
```

The second function performs one multiplication and one add operation to generate a random number. The first function returns 31 bits of this result.

3.2.3.3. Combined Multiple Recursive Generator

The CMRG returns a random number by using two sub-functions. The source codes for the CMRG are described as follows [2]:

```
/*** Source Code from SPRNG [2] ***/
int CMRG::get_rn_int()
   {advance_cmrg();  return ((state+(s0<<32))&MULT_MASK)>>33;}
void CMRG::advance_cmrg()
   { unsigned LONG64 p; advance_state(); p = a0*s0 + a4*s4; p = (p>>31) + (p&0x7fffffff);
   if(p&0x80000000) p = (p+1)&0x7fffffff;  s4=s3; s3=s2; s2=s1; s1=s0; s0=p;}
void CMRG::advance_state()
   {state = state*multiplier + prime;}
```

The advance_state() function has the same functionality as with the 64 bit LCG. The advance_cmrg() function calls the 64 bit LCG operation. Additionally, it performs two multiplications and one add operation with 4 variables to produce an output. The output produces two modified results. One is the value obtained by the shift-right operation, and the other one is the value from the mask operation. Adding the two values produce a new output. If the 32nd bit of the output is '1', the output is modified by adding '1' and a mask operation. In SPRNG the output goes to a 5 deep FIFO register. For HASPRNG, the FIFO depth is '4', since the implementation employs forwarding. Finally, the get_rn_int() function returns a random number by one add, one masking operation, and two shifting operations.

3.2.3.4. Multiplicative Lagged Fibonacci Generator

The MLFG has three functions to generate integer random numbers. The three functions are described as follows [2]:

```
/*** Source Code from SPRNG [2] ***/
int MLFG::get_rn_int()
   { advance_state(); return lags[hptr]>>33;}
void MLFG::advance_state()
   { int lv = lval, kv = kval;  int lptr;  hptr--;  if(hptr < 0)  hptr = lv-1; lptr = hptr + kv;
  if (lptr>=lv)  lptr -= lv;  multiply(lags[hptr],lags[lptr],lags[hptr]); }
#define multiply(a,b,c) {c = (a)*(b); c &= MASK64;}
```

The first function controls generation of a random number. The second function is used to manipulate lag factors. The multiply function multiplies two 64 bit values, and keeps the lower 64 bits. The get_rn_int() function returns a 31 bit random number. The advance_state() function calls the multiply() function which multiplies the data at the higher pointer address by the data at the lower pointer address. The 64 bit product is stored back at the higher pointer address. The variables for the higher pointer (hptr), length (lval), lag (kval), and data (lags) are member variables in the MLFG class. By changing the member variables in the functions, the functions can access different data inside the MLFG class, since the functions are also member functions in the MLFG class. When the get_rn_int() function calls the advance_state() function, the higher pointer is reduced by '1' to indicate the next data. The higher pointer is initialized when it goes a negative value. The lower pointer value is determined by the higher pointer value and the lag value, and is reset when it is greater or equal than the length value.

3.2.3.5. Prime Modulus Linear Congruential Generator

The PMLCG employs two functions in order to generate and return a random number as follows [2]:

```
/*** Source Code from SPRNG [2] ***/
int PMLCG::get_rn_int()
   { iterate();   return (int) (x>>30); }
void PMLCG::iterate()
   { unsigned LONG64 x0, x1, x3, ul, uh, vl, vh;
   #define MULT_MASK1  0x7fffffffU
   #define MULT_MASK2 0x3fffffffU
   #define MULT_MASK3 0x1fffffffffffffffUL
   #define MULT_MASK4 0x2000000000000000UL
   ul = mult&MULT_MASK1;  uh = (mult>>31)&MULT_MASK2;
    vl = x&MULT_MASK1;  vh = (x>>31)&MULT_MASK2;
```

```
 x0 = ul*vl;
x1 = ul*vh + uh*vl + (x0>>31);
 x0 &= MULT_MASK1;
x3 = ((uh*vh)<<1) + (x1>>30);
 x0 |= (x1&MULT_MASK2)<<31;
 x = (x0+x3);
if(x&MULT_MASK4)
{ x &= MULT_MASK3;  x += 1;  if(x == MULT_MASK4)  x = 1; } }
```

The variables for the multiplier (mult), and the seed (x) coefficients reside in the PMLCG class. By the iterate() function, the variables can be accessed and changed. In the iterate() function, the initial mult and seed values are divided into parts. The function performs the four multiply operations and data manipulations using the mask operations. We can expect performance improvement at this point since those four multiply operations can occur at once in FPGA. The if statement in the function performs the prime modulus operation, composed of one mask operation and add operation, and the data check operation to prevent the generator from producing '0' as a random number. The data after the data check operation is returned by the get_rn_int() function as a random number.

3.2.4. Generator Implementations

The random number generator implementations are described in terms of architectures and VHDL descriptions in this section. The architectures section provides the dataflow of the generators and the VHDL implementation section illustrates how to implement the architectures with VHDL.

3.2.4.1. Architectures

In order to design hardware with VHDL, we have to devise appropriate and effective architectures with consideration of performance and hardware usage. In the architecture designs, the original SPRNG characteristic equations are transformed to different forms to improve efficiency of the implementations as necessary.

3.2.4.1.1. 48 bit Linear Congruential Generator

The 48 bit LCG is characterized by the following equations [2]:

$$Z(n) = a \times Z(n-1) + p \text{ (Mod } 2^{48})$$

where Z(n) is a random number on the generator, *p* is the prime number and *a* is the multiplier coefficient. The equation does not allow us to use a pipelined multiplier for high performance. Using a pipelined multiplier for the equation results in reduced performance, since the current random number accesses the previous random number. The 48 bit inputs are too large for a fast, non-pipelined multiplier resulting in slow clock speed due to circuit delays. To overcome these disadvantages, the equation should be transformed into another form for the implementation. When the equation is observed carefully, a recursive relation for random number generation is revealed. The equation transformation is achieved from the following process:

$Z(1) = a \times Z(0) + p \text{ [Mod } 2^{48}]$

$Z(2) = a \times Z(1) + p \text{ [Mod } 2^{48}] = a \times (a\times Z(0) + p) + p \text{ [Mod } 2^{48}] = a^2\times Z(0) + p\,(a + 1) \text{ [Mod } 2^{48}]$

$Z(3) = a\times Z(2) + p\text{[Mod } 2^{48}] = a\times(a^2\times Z(0)+p(a+1))+p\text{[Mod } 2^{48}] = a^3\times Z(0) + p\,(a^2+a+1)\text{[Mod } 2^{48}]$

Likewise,

$Z(8) = a^8 \times Z(0) + p \times (a^7 + a^6 + a^5 + a^4 + a^3 + a^2 + a^1 + 1)$ [Mod 2^{48}]

$Z(8) = \alpha \times Z(0) + p \times \beta$[Mod 2^{48}]$=\alpha \times Z(0) + \gamma$[Mod 2^{48}]($\alpha = a^8$, $\beta = a^7+a^6+a^5+a^4+a^3+a^2+a^1+1$, $\gamma = p \times \beta$)

where Z(0) is an initial seed, which can be any positive value. α, and β are constants which are obtained from a multiplier coefficient. The γ is also a constant obtained from the prime number multiplied by β. The changed equation produces identical functional results to the previous equation. The new equation allows us to use seven-stage pipelined multipliers for the LCG design. The new equation is employed for the LCG implementation. The hardware architecture for the LCG is designed to generate 8-ahead random numbers according to current inputs. Figure 5 describes the 48 bit LCG hardware architecture. In the LCG implementation, two multipliers are employed. There are three parts of the LCG implementation. At first, the LCG module obtains required coefficients (α, and β), and the first 7 random numbers. The α and β coefficients are obtained from the left multiplier, and the γ and the first 7 random numbers are obtained from the right one. They are displayed in the shaded regions of Figure 5. In the second part, once the LCG obtains the initial coefficient values, it spawns the first 7 random numbers prior to the actual operation. In the last part, the LCG generates a random number using the α and γ coefficients which satisfy the new equation. The 48 bit LCG generates a random number every clock cycle.

3.2.4.1.2. 64 bit Linear Congruential Generator

The 64 bit LCG hardware architecture is the same as the 48 bit LCG. Its implementation also follows the equation of the 48 bit LCG, but with a larger Mod size. The Mod size should be changed to 2^{64}. The multipliers are changed to 64 bit multipliers instead of 48 bit multipliers. The 64 bit LCG generates a random number every clock cycle.

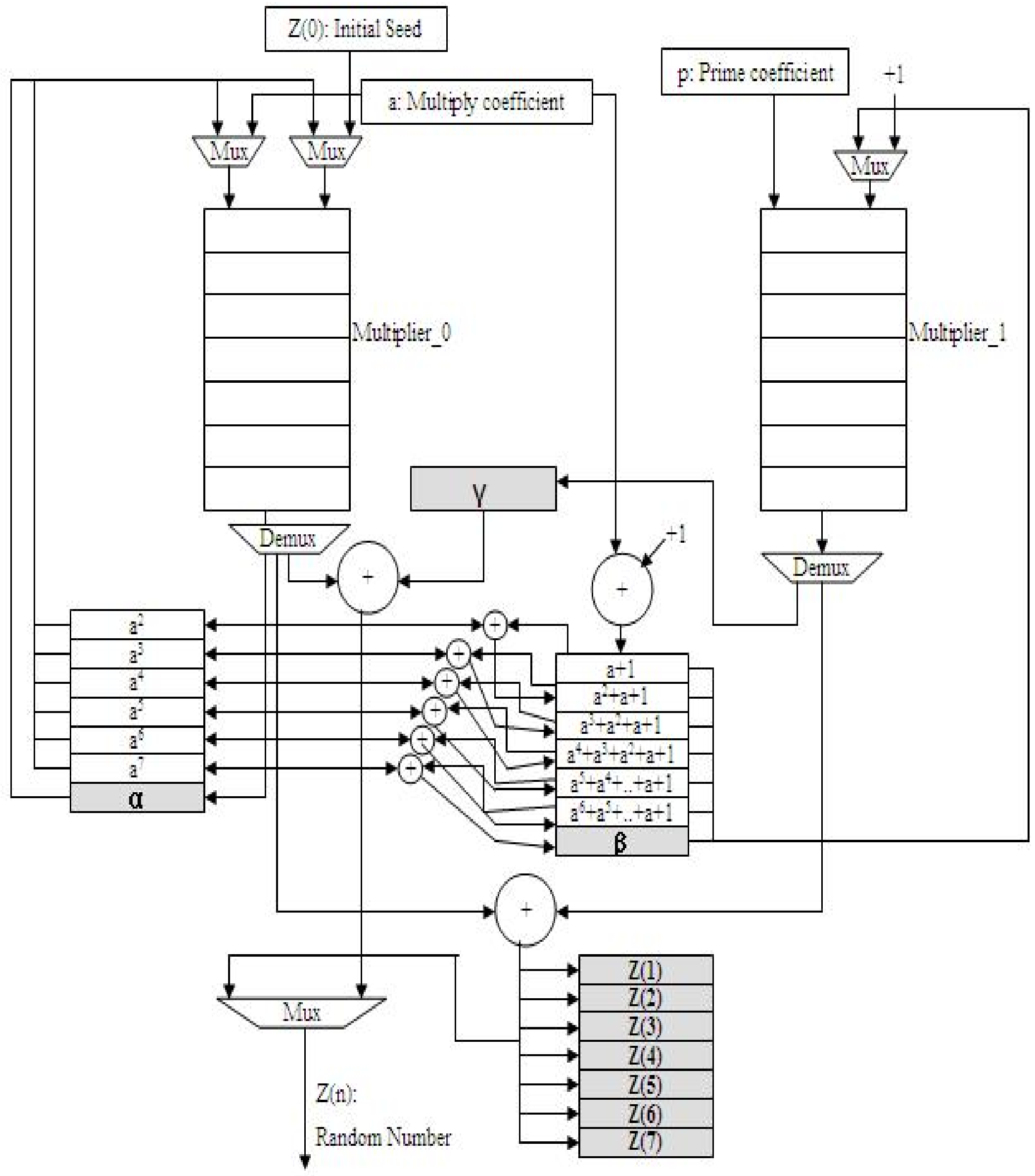


Figure 5. 48/64 bit LCG Architecture

3.2.4.1.3. Combined Multiple Recursive Generator

The CMRG in SPRNG is characterized by the following equations [2]:

$$Z(n) = X(n) + Y(n) \times 2^{32} \ (\mathrm{Mod}\ 2^{64})$$

$$Y(n) = 107374182 \times Y(n-1) + 104480 \times Y(n-5) \ (\mathrm{Mod}\ 2^{31}-1)$$

where X(n) is generated by 64 bit LCG and Z(n) is the resulting random number. The Y(n) multiplied by 2^{32} represents a new 64 bit random number. The equation is modified slightly as follows:

$$Z(n) = X'(n) + Y(n)$$

where X'(n) is the upper 32 bits of X(n). The new equation produces the identical results with the ones from the previous equation, since Z(n) returns the higher 31 bits for a random number.

Figure 6 shows the CMRG hardware architecture. The architecture has two parts, each generating a partial result. The first part is a 64 bit LCG, and the second part is the generator having two lag factors. The left part's architecture is the same as the LCG hardware architecture above. The right part is composed of two multipliers, four deep FIFO registers, and some combinational logics. The FIFO registers have the initial values with '1'. The two-stage pipelined multipliers generate results every other clock cycle. Hence, the CMRG random number generation produces a result every two clock cycles. On the first operation, the value '1' is given to one of the inputs of the left multiplier. The output from the FIFO registers is given to one of the inputs of the right multiplier. After the first operation, the multiplexer on the left multiplier changes the input from the value '1' to the result obtained from the operation.

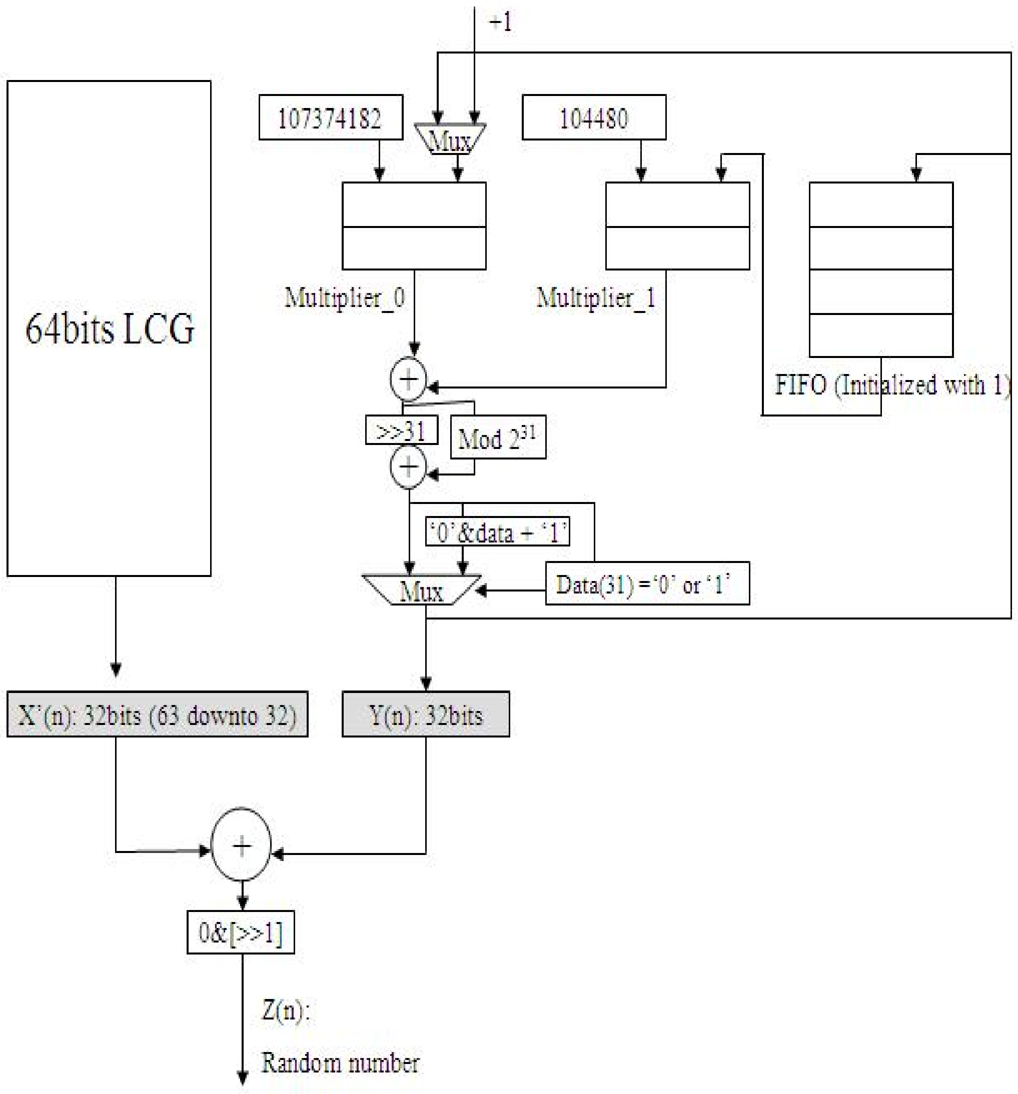


Figure 6. CMRG Architecture

The two multiplier output data are added and a modulus operation is applied. For the Mod operator, the data is shifted right by 31 bits, and it is added to the lower 31 bits of the data before the shifting. In the shift operation, the truncation of bits is employed. The data after the Mod process fans out three ways. The first one goes to the left multiplier input port on the figure in order to save clock cycle latency. The second one goes to the FIFO registers. According to this forwarding scheme, the FIFO registers size is reduced from 5 to 4. The third one goes to the final adder, which adds the data to the data generated by the 64 bit LCG module. There are some conditions we have to consider on this final add operation. The 64 bit LCG described above generates a random number every clock cycle. The right part generator generates a random number every other clock cycle. The 64 bit LCG needs around 50 clock cycles for the initialization. In order to synchronize these conditions, the CMRG employs an enable signal, and a clock divider. The enable is used to enable the right part generator right after the LCG initialization process is finished, and a clock divider is used to decrease the LCG generation speed so that it matches the right part's generation speed. The data after the final add and shift operations represents a random number. The CMRG generates a random number every other clock cycle.

3.2.4.1.4. Multiplicative Lagged Fibonacci Generator

The MLFG satisfies the following equation [2]:

$$Z(n) = Z(n-k) \times Z(n-l) \ (\mathrm{Mod}\ 2^{64})$$

where *k* and *l* are time lags and Z(n) is the resulting random number [2].

Figure 7 describes the MLFG hardware architecture. The MLFG uses 3 dual port RAMs. The first one (Mem_load) is used for data loading from one of the PowerPCs on the board. The dual port RAM has 32 bit data input ports and 64 bit data output ports. The PowerPC gives 32 bit data to the dual port RAM, and a controller (Controller 0) reads 64 bit data from the dual port RAM. The other two dual port RAMs (Mem_bank) are used to hold the previous results (Z(n-l), and Z(n-k)). After data loading is done in the first dual port RAM, an enable signal activates a controller module (Controller 0), which loads 64 bit data to the two dual port RAMs. Once the data loadings are complete, the controller (Controller 0) activates another controller (Controller 1), which reads data from the two dual port RAMs. The two values read by the controller (Controller 1) from the dual port RAMs go to the multiplier. The multiplier output goes back to a controller (Controller 2) in order to store the data to the two dual port RAMs. The data stored are read again later by the controller (Controller 1). This multiplier output also represents the resulting random number after truncating the higher 31 bits out of the 64 bit product. The MLFG generator produces a random number every clock cycle.

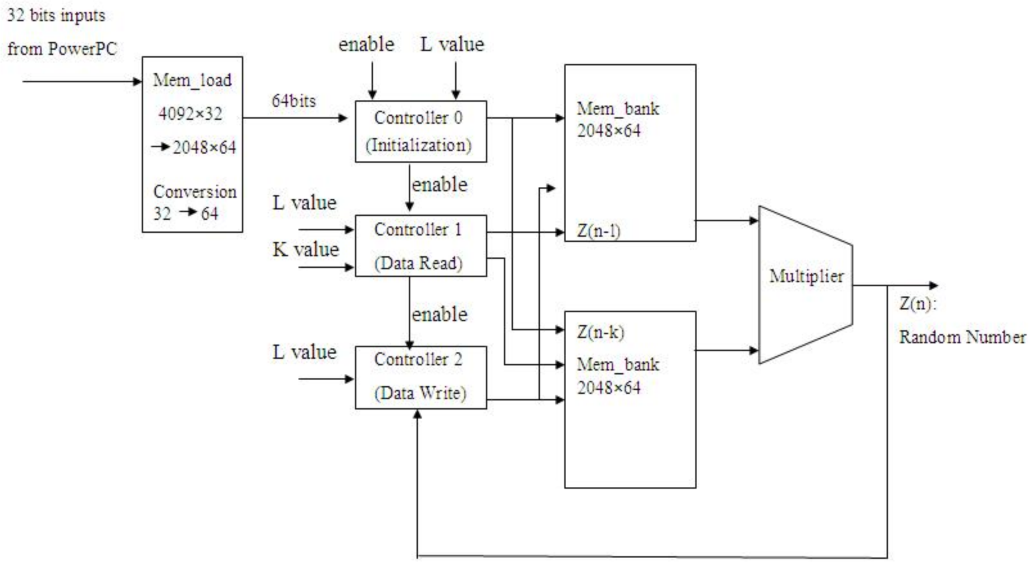


Figure 7. MLFG architecture

3.2.4.1.5. Prime Modulus Linear Congruential Generator

The PMLCG satisfies the following equation [2]:

$$Z(n) = a \times Z(n-1) \times 2^{32} \text{ (Mod } 2^{61}-1)$$

where a is a multiplier parameter and Z(n) is the resulting random number [2]. Figure 8 shows the PMLCG hardware architecture. In the PMLCG hardware architecture, four two-stage pipelined 32bit multipliers are employed. The four multipliers operate concurrently. Also, a multiplexer is employed to save clock cycles. On the first operation, the initial seed and the initial multiplier coefficient are divided by the lower 31 bits and higher 30 bits. The value of '0' is attached to the lower 31 bit data at the most significant bit position. Likewise, the value of "00" is attached to the higher 30 bit data at the most significant bit positions. The last part of the generator hardware implementation is described by the IF-condition black box in the figure. The IF-condition black box performs the modulus and data check operations. When the 62nd bit of the data is '1', the data is modified by adding '1' after the 62 bit mask operation. If the modified data is 2^{61}, the data is changed to the value '1' in order to prevent the generator from producing '0' as a random number [1]. The operation is represented by the following pseudo code [2]:

```
 if (seed(61))
{ seed &= Mask61bits;  seed += 1;  if (seed = $2^{61}$)  seed = 1; }
```

The final data through the IF-condition black box fans out in two directions. The first is fed into one of the inputs of the multiplexer in order to generate the next random number. The second represents the random number. After the first operation, the multiplexer always select the data instead of the initial seed. The PMLCG generates a random number every other clock cycle.

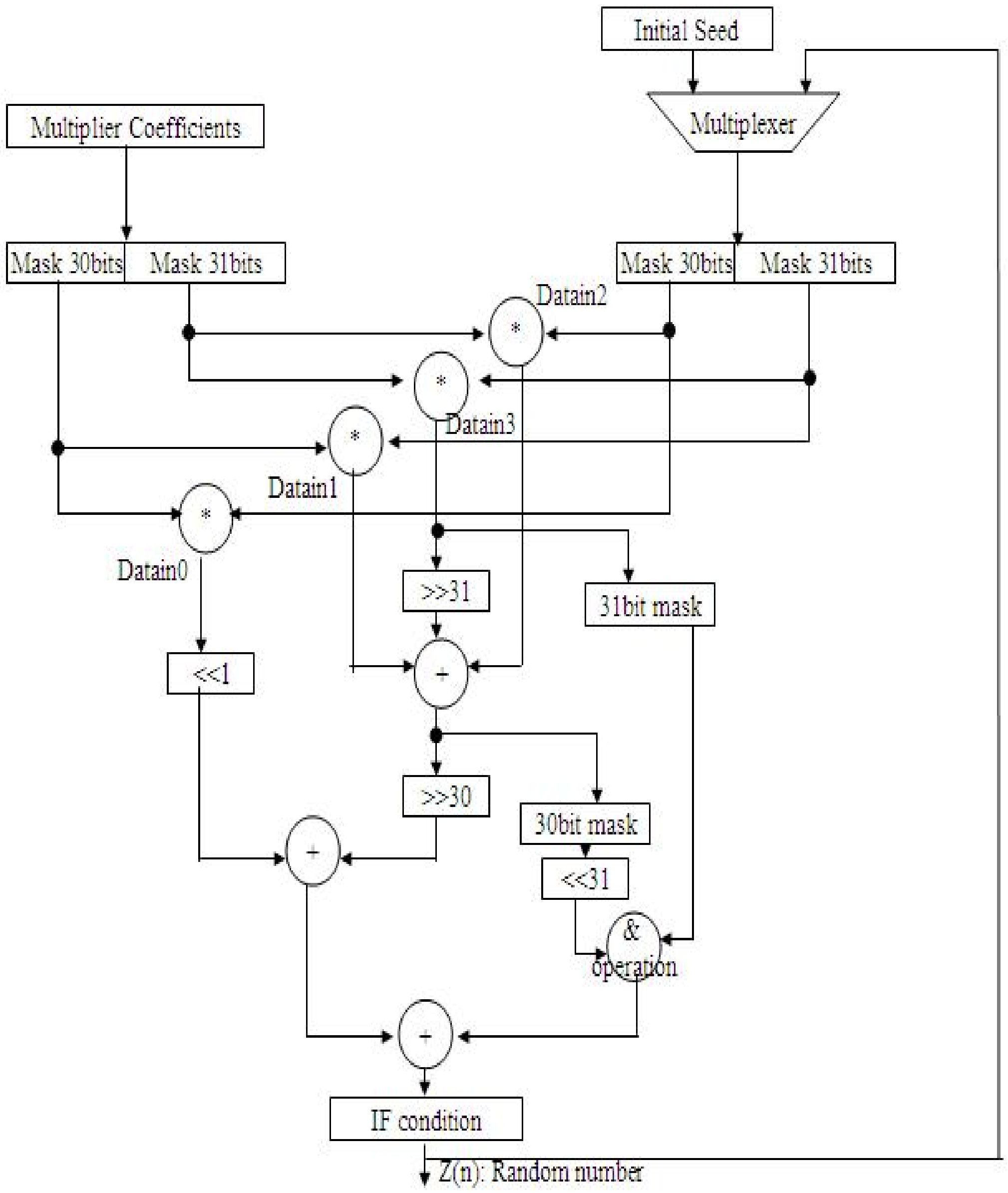


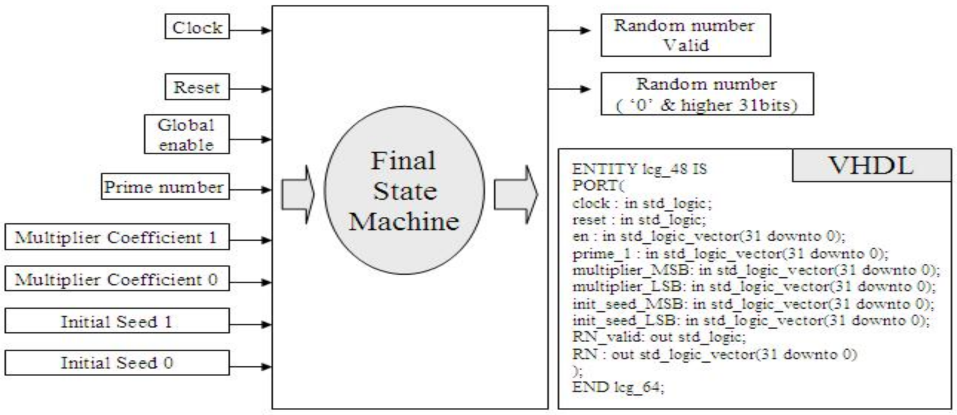
Figure 8. PMLCG Architecture

3.2.4.2. VHDL Designs

The modularity concept is used for implementation. When a design is getting complicated, a design considering modularity is one of the best approaches. We can fix bugs with less effort on a design considered modularity. All the generators designs except the LCGs are designed with the modularity concept. The LCGs are designed with a FSM (Finite State Machine), which has 6 bits for the number of the states. A FSM is appropriate to manipulate signal transactions in a module. LCGs use deeply pipelined multipliers in order to get high performance.

3.2.4.2.1. 48 bit Linear Congruential Generator with Prime Addend

For the 48 bit LCG implementation, the 6 bit FSM is employed to handle the initialization process efficiently. The two 48 bit multipliers are used for the implementation. Figure 9 shows the entity declaration for the LCG. The 8 inputs and 2 outputs are declared for the entity with consideration of the block memory data transfer. The block memory data transfer is essential for the programming interface and verification implementation. We explore the block memory transfer mechanism in Chapter 4 and Chapter 5. The RN_valid output is connected to a dual port RAM to store the random number in the dual port RAM when the random number is valid.



```
ENTITY lcg_48 IS
PORT(
clock : in std_logic;
reset : in std_logic;
en : in std_logic_vector(31 downto 0);
prime_1 : in std_logic_vector(31 downto 0);
multiplier_MSB: in std_logic_vector(31 downto 0);
multiplier_LSB: in std_logic_vector(31 downto 0);
init_seed_MSB: in std_logic_vector(31 downto 0);
init_seed_LSB: in std_logic_vector(31 downto 0);
RN_valid: out std_logic;
RN : out std_logic_vector(31 downto 0)
);
END lcg_64;
```

Figure 9. 48 bit LCG Entity

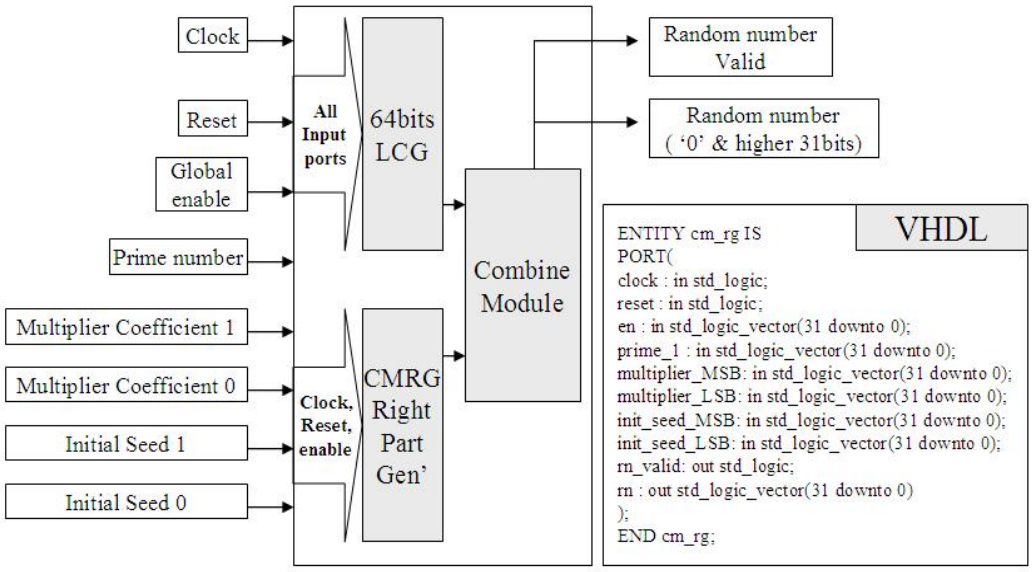
The en input is used to enable, and stop the module for the block memory data transfer. The clock input is used to activate the module, and the reset input is used to reset the module’s operation. The rest of the inputs are used to get the coefficient values given by SPRNG. The multiplier coefficient inputs and initial seed inputs are divided into parts to get the higher 16 bits, and lower 32 bits out of 48 bit data. The multipliers are made by the Xilinx-COREGEN tool, and have a seven-staged pipeline architecture with an enable bit. The multipliers preserve the internal data when the multiplier operation is stopped.

3.2.4.2.2. 64 bit Linear Congruential Generator with Prime Addend

The 64 bit LCG VHDL implementation is similar to the 48 bit LCG. Two 64 bit multipliers are used for the implementation instead of the 48 bit multipliers. Only data size modifications are necessary from the 48 bit LCG. For the 64 bit LCG implementation, 48 bit data are changed to 64 bit data, and 96 bit data from a multiplier output are changed to 128 bit data.

3.2.4.2.3. Combined Multiple Recursive Generator

The CMRG consists of 5 sub modules. The 5 sub modules are represented by the 64 bit LCG, the right generator having 2 sub modules, and a combine module. The CMRG top entity is described by Figure 10. In the CMRG VHDL description, 4 multipliers are generated by the CORGEN tool. The two multipliers have 64 bit data inputs, and the other two multipliers have 32 bit data inputs. The CMRG entity description is exactly same as the 64 bit LCG, since the interface for the CMRG only depends on the 64 bit LCG. When you observe the CMRG hardware architecture, the right generator coefficient values are decided by constant values. It is not necessary to get any coefficient value from outside the module. The 64 bit LCG implementation is the same as the one described above.



```
ENTITY cm_rg IS
PORT(
clock : in std_logic;
reset : in std_logic;
en : in std_logic_vector(31 downto 0);
prime_1 : in std_logic_vector(31 downto 0);
multiplier_MSB: in std_logic_vector(31 downto 0);
multiplier_LSB: in std_logic_vector(31 downto 0);
init_seed_MSB: in std_logic_vector(31 downto 0);
init_seed_LSB: in std_logic_vector(31 downto 0);
rn_valid: out std_logic;
rn : out std_logic_vector(31 downto 0)
);
END cm_rg;
```

Figure 10. CMRG Entity

However, the 64 bit LCG module is activated by a slower clock from a clock divider output. The clock divider is instantiated in the CMRG top module inside a process ( ) statement. The right generator has two sub modules, which are used for FIFO registers operation and the modulus operation. The combine module is used to add the random numbers from the 64 bit LCG and the right generator.

3.2.4.2.4. Multiplicative Lagged Fibonacci Generator

The MLFG consists of 6 sub modules. 3 sub modules are used for memory, and 3 sub modules are used for controlling address access to memory modules. All the memory modules are implemented by dual port RAMs. One port is used for data read, and the other port is used to write data into the memory. All three memory modules have a 14 bit address. One bit of the address can indicate to one byte of data. The memory entity modules are characterized by port enable bits, byte enable bits, and write enable bits. One of the memory modules is used for initial seeds data loading. In this case, data conversion occurs. The 32 bit data read from the PowerPC are stored as 64 bit data. This transform is possible by implementing a multiplexer having an output as port-enable bits and a select bit as the third bit of input address data. The mechanism is explained by Figure 11. In the figure, the port a is used for reading data, and the port b is used for writing data. For example, when the first 32 bit data is loaded and the third bit of the input address is ‘0’, 4 bytes of the port enable bits are set to high. When the second 32 bit data is loaded and the third bit of the input address is ‘1’, the other 4 bytes of the port enable bits are set to high. The mechanism can let the entity have 32 bit input data, and 64 bit output data. The other two memory modules are used to read and store back data from the multiplier output on the MLFG execution. One port is used for read operation, and the other port is used for write back operation. The three types of controller modules have different functionalities. All the controllers have their own address generators inside, and two types of enable signals. One is used for enabling a module locally, and the other is used for enabling a module globally. In the former case, the enable signal comes from the module’s local operation, and in the later case, the enable signal comes from PowerPC.

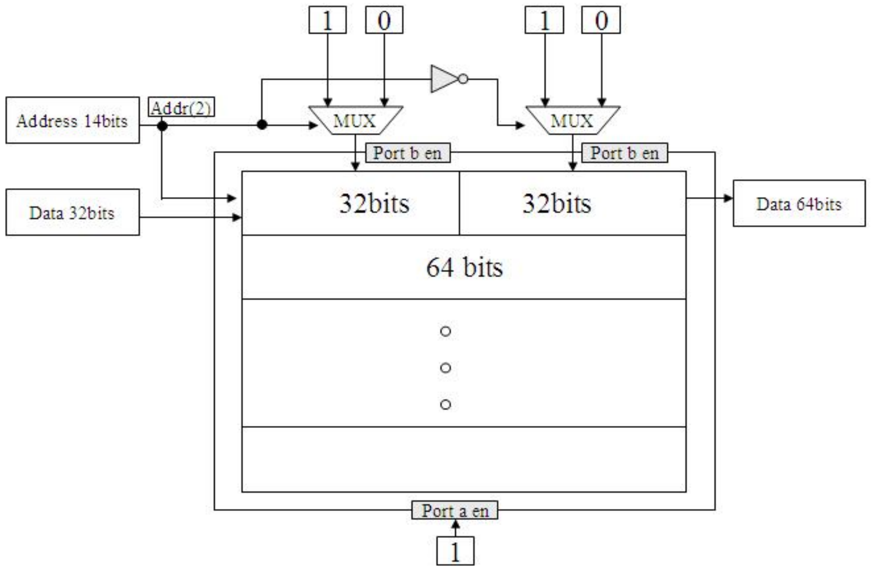
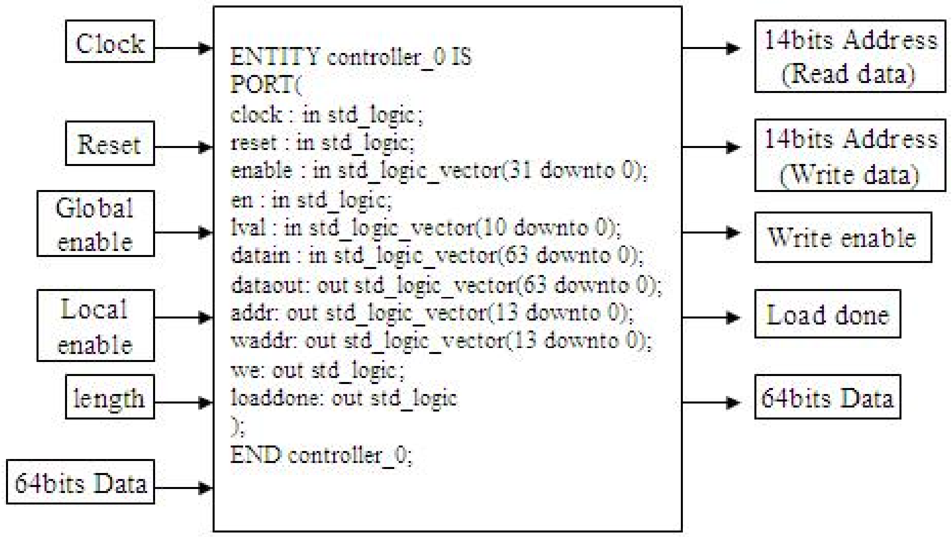


Figure 11. Architecture for Data Conversion

The first controller is used to load initial seed data from one of the memory modules to the other memory modules. The controller entity is characterized by 6 inputs and 5 outputs shown in Figure 12. When the initial seeds are loaded in one of the memory modules, and the select bit from the second controller (Controller 1) is "LOW" (Figure 13). The controller transfers the initial seed data from the memory module to the other two memory modules. According to the length value, the controller generates appropriate addresses (Write data), Write enable signal, Load done signal, and data in order to store the data to the other two memory modules. When this process is complete, the controller sets the Load done signal to '1' to enable another controller module.

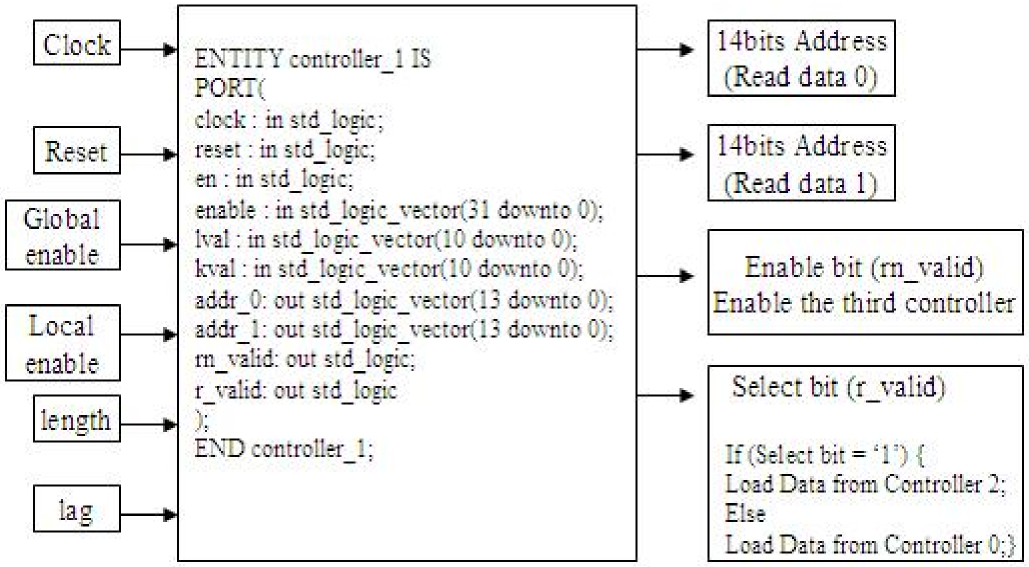
The second controller is used to read data in order to give data to the multiplier. The controller entity module is characterized by 6 inputs and 4 outputs as shown in Figure 13. Controller 1 accesses 64 bit data from two memory modules in order to feed the data to the multiplier. While it gives the data to the multiplier, it generates two signal types. The first one is used to enable the other controller module, and the other is used to select the data between the two sources. The data from the first controller is initial seeds, and the data from the third controller is the multiplier output (the random number). According to the select bit, the selected data are stored to the two memory modules.

The last controller module is used to write back the multiplier output data at appropriate memory address. The controller entity module is characterized by 6 inputs and 4 outputs as shown in Figure 14. The controller is used to load the multiplier output data to the two memory modules for operations. The write enable bit goes to the two memory module write enable ports through one multiplexer as in Figure 15. All the sub modules described above are connected in the top module of the MLFG.



```
ENTITY controller_0 IS
PORT(
clock : in std_logic;
reset : in std_logic;
enable : in std_logic_vector(31 downto 0);
en : in std_logic;
lval : in std_logic_vector(10 downto 0);
datain : in std_logic_vector(63 downto 0);
dataout: out std_logic_vector(63 downto 0);
addr: out std_logic_vector(13 downto 0);
waddr: out std_logic_vector(13 downto 0);
we: out std_logic;
loaddone: out std_logic
);
END controller_0;
```

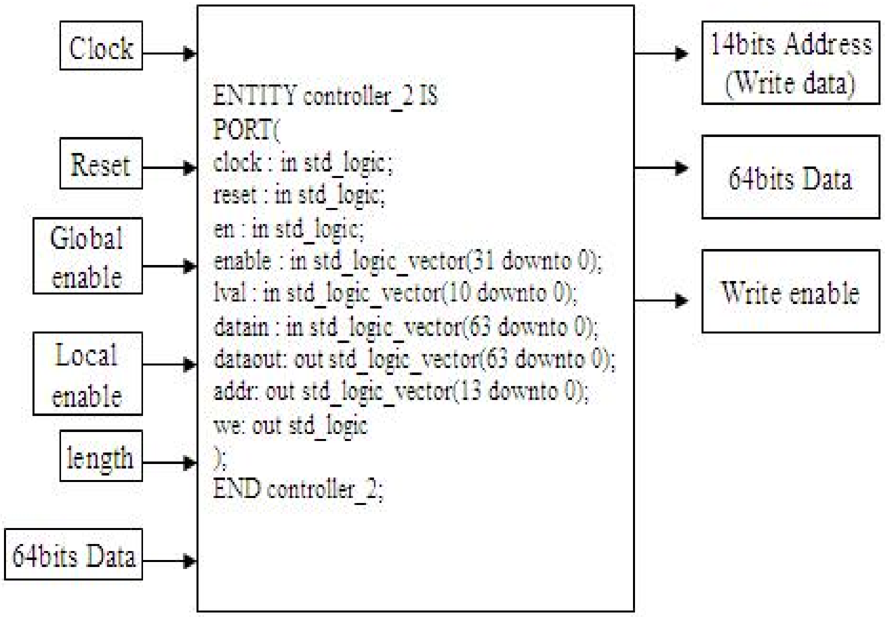
Figure 12. The Controller Entity for Loading Initial Seeds



```
ENTITY controller_1 IS
PORT(
clock : in std_logic;
reset : in std_logic;
en : in std_logic;
enable : in std_logic_vector(31 downto 0);
lval : in std_logic_vector(10 downto 0);
kval : in std_logic_vector(10 downto 0);
addr_0: out std_logic_vector(13 downto 0);
addr_1: out std_logic_vector(13 downto 0);
rn_valid: out std_logic;
r_valid: out std_logic
);
END controller_1;
```

Clock
Reset
Global enable
Local enable
length
lag
14bits Address (Read data 0)
14bits Address (Read data 1)
Enable bit (rn_valid) Enable the third controller
Select bit (r_valid)
If (Select bit = '1') {
Load Data from Controller 2;
Else
Load Data from Controller 0;}

Figure 13. The Controller Entity for Reading Data



```
ENTITY controller_2 IS
PORT(
clock : in std_logic;
reset : in std_logic;
en : in std_logic;
enable : in std_logic_vector(31 downto 0);
lval : in std_logic_vector(10 downto 0);
datain : in std_logic_vector(63 downto 0);
dataout: out std_logic_vector(63 downto 0);
addr: out std_logic_vector(13 downto 0);
we: out std_logic
);
END controller_2;
```

Figure 14. The Controller Entity for Writing Back Data

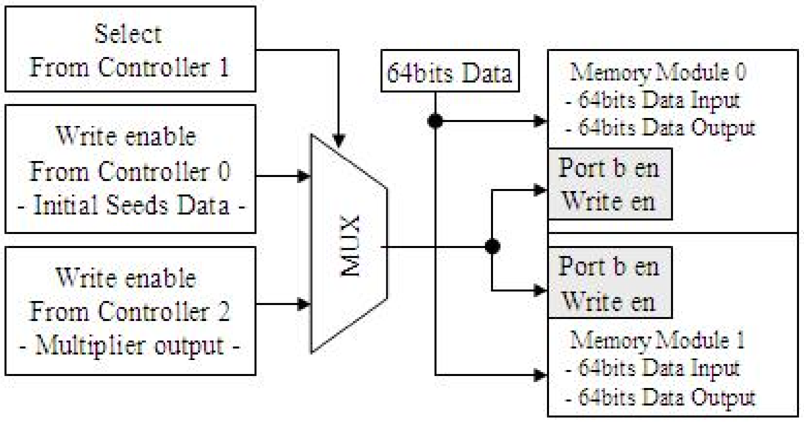
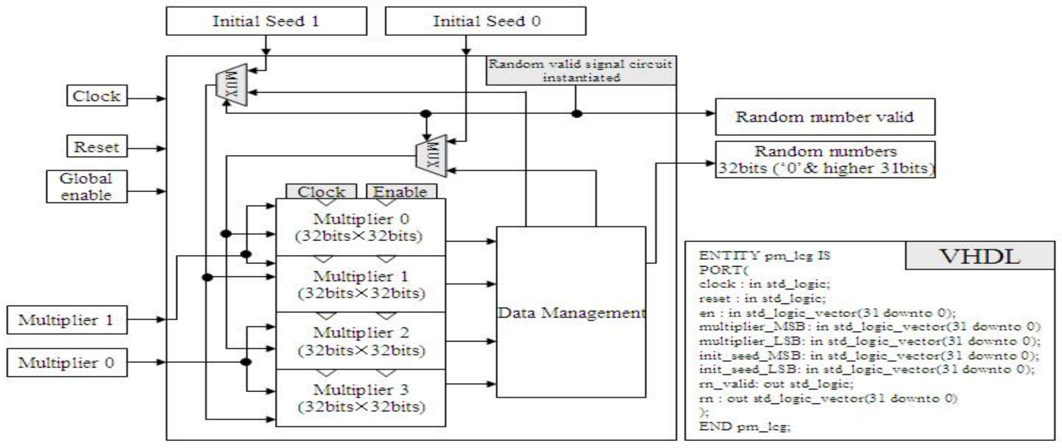


Figure 15. Data Selection

3.2.4.2.5. Prime Modulus Linear Congruential Generator

The PMLCG top entity is composed of 4 32bit multipliers, an instantiated random valid signal generator circuit, and one data management module. The PMLCG top module is described by Figure 16. The PMLCG top entity has 7 input ports, and two output ports. The clock input is used to activate the module, and the reset signal is used to reset the module. The enable input port is able to activate and stop the PMLCG operation. The initial seed input ports are used to give initial seed values to one of the multipliers inputs. The multiplier coefficient input ports go to the other port of the multiplier inputs. After the first operation, the data through the data management module are given to the input ports of the multipliers. The shaded multiplexers select data between the two by a control bit generated by the random valid signal circuit. The data management module entity has 4 input ports for receiving data and 3 output ports for producing data. The data management module employs combinational logic in order to save clock cycles. The data management module takes three responsibilities. The data management module arranges the data from the 4 data inputs, performs the Prime-Modulus (Mod 2^{61}-1) operation, and checks the data. The data check operation is necessary because the data after the modulus operation is '0' if the data before the modulus operation equals to "2^{61}-1". The random number is represented by the one of the output ports in the data management module.


Initial Seed 1
Initial Seed 0
Random valid signal circuit instantiated
Clock
Reset
Global enable
MUX
MUX
Random number valid
Random numbers 32bits ('0' & higher 31bits)
Clock
Enable
Multiplier 0 (32bits×32bits)
Multiplier 1 (32bits×32bits)
Multiplier 2 (32bits×32bits)
Multiplier 3 (32bits×32bits)
Data Management
Multiplier 1
Multiplier 0
VHDL

```
ENTITY pm_lcg IS
PORT(
clock : in std_logic;
reset : in std_logic;
en : in std_logic_vector(31 downto 0);
multiplier_MSB : in std_logic_vector(31 downto 0)
multiplier_LSB: in std_logic_vector(31 downto 0);
init_seed_MSB: in std_logic_vector(31 downto 0);
init_seed_LSB: in std_logic_vector(31 downto 0);
rn_valid: out std_logic;
rn : out std_logic_vector(31 downto 0)
);
END pm_lcg;
```

Figure 16. PMLCG Entity

3.3. Summary

In prior work [3, 4, 20], the SPRNG Modified LFG generator was implemented. The remaining five generators in SPRNG are implemented for FPGAs. The software source code of SPRNG was described for the implementations. Some portions of characteristic equations in SPRNG have been modified for effective hardware implementations. The LCGs and MLFG are designed to generate a random number every clock cycle, and the CMRG and PMLCG are designed to generate a random number every other clock cycle.

Chapter 4. Verification

This chapter illustrates the rigorous verification platform which performs bitwise checks on the random numbers generated by HASPRNG. The verification is performed using an embedded IBM PowerPC 405 processor in the XC2VP30 FPGA that compares results from HASPRNG with the execution of SPRNG. The PowerPC includes data and instruction cache, embedded memory management unit, execution unit, and on-chip memory interfaces [26]. The verification flow section provides an overview of the verification process, and the hardware and software verification platforms are described in this chapter.

4.1. Verification Flow

For HASPRNG verification, the SPRNG 4.0 library was ported to the PowerPC platform using the powerpc-eabi-gcc compiler. That makes the PowerPC able to easily bitwise compare the results from both SPRNG and HASPRNG. The verification flow is described by Figure 17. The main C++ source code is compiled by the powerpc-eabi-gcc compiler with linking C++ libraries (-stdc++, and –supc++), hardware library and the SPRNG library. When the compilation is complete, the compiler generates an executable file. The executable file is too large to load the file into the block memory, since the executable file accesses the SPRNG library.

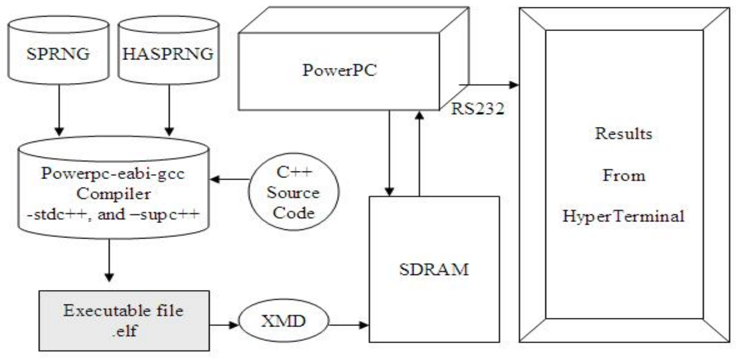


Figure 17. Verification Flow

The XMD is used to load the executable file to SDRAM, and execute the file using the PowerPC. The PowerPC checks the data validation to ensure each HASPRNG generator provides equivalent behavior as with SPRNG and shows the verification results through the HyperTerminal.

4.2. Hardware Platform for Verification

Seamless data transfer is one of the most important considerations for the verification of HASPRNG. We discuss architectural support to simplify data transfer so the PowerPC reads data continuously and can verify HASPRNG using SPRNG. The implementation subsection describes how we realize the architecture using VHDL.

4.2.1. Architecture

The hardware architecture should provide the environment to let the PowerPC read each random number from HASPRNG in order to compare the data to the corresponding one from SPRNG. For the task, the hardware architecture employs a large dual port RAM (128KByte). The data from HASPRNG resides in the dual port RAM, and is read by the PowerPC. When the dual port RAM is full of data from the HASPRNG operation, the HASPRNG module operation pauses while keeping the next random number. The architecture is designed for a HASPRNG test module to be able to stop locally to simplify the control. A PowerPC command can activate the module again, once the PowerPC completes the data verification. Figure 18 describes the verification hardware architecture. The local controller (Controller 0) tells the master controller (Controller 1) when the dual port RAM is full of data, and then the local controller initializes the full flag for the next stop operation. The master controller stops and enables a random number generator according to the signals from both the local controller and the PowerPC.

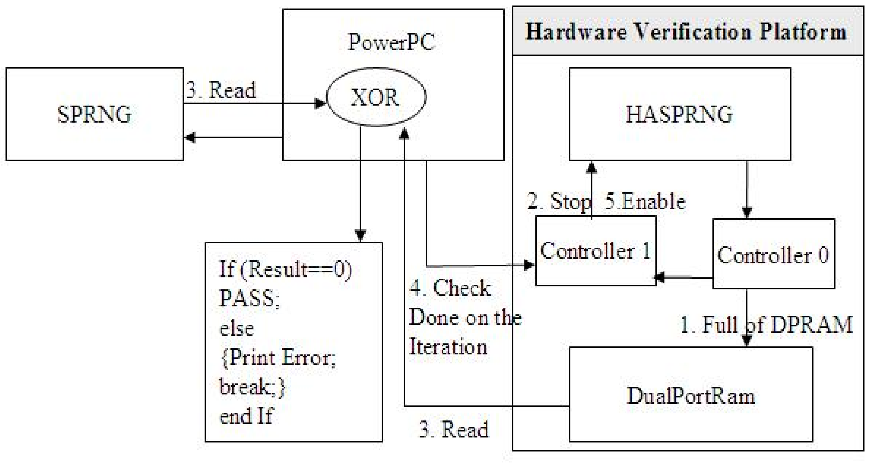


Figure 18. Hardware Platform for Verification

When the master controller receives the full flag signal from the local controller, the master controller stops the HASPRNG operation. The PowerPC reads data from both SPRNG and HASPRNG, and verifies the HASPRNG data by employing a bitwise XOR function. When all the 32K random numbers are checked, the PowerPC allows the master controller to resume HASPRNG operations.

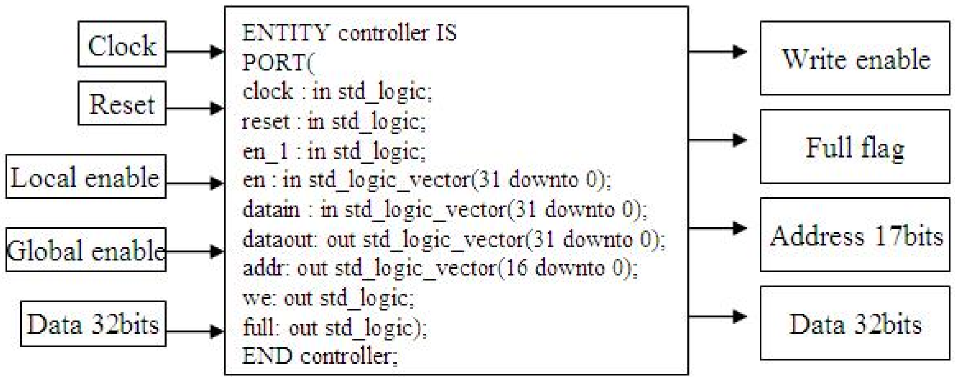
4.2.2. Implementation

The hardware platform employs the SDRAM for storing and executing an executable file. Some parameter sets of DCM should be modified in order to use the SDRAM [7]. The PowerPC is set to 100 MHz for both PLB and OPB for the verification of all the generators except MLFG. In the MLFG, the PLB is set to 200MHz, and the OPB is set to 66.7MHz, since the MLFG can not run at 100MHz.

The hardware verification platform employs one 4 x 32KByte block memory module and two controllers. The block data transfer (128KByte) is performed by the three modules. All the HASPRNG modules store the 32K random numbers into the 128KByte block memory. When

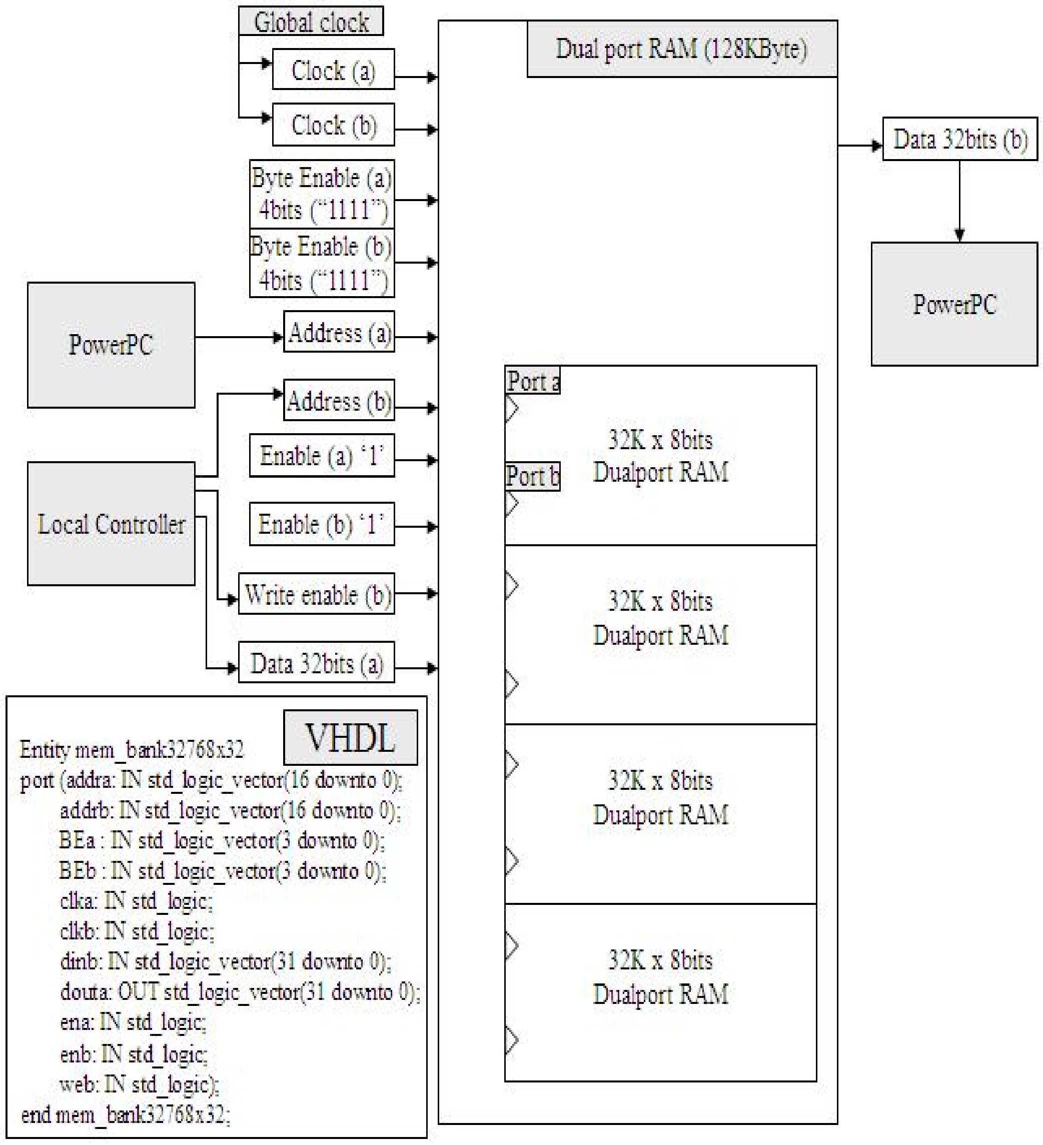
HASPRNG stores the random numbers data in the dual port RAM, the data goes through the local controller first.

Two tasks are assigned to the local controller. The first task is to load the data at the correct address in the dual port RAM. The other task is to tell the master controller that the memory is full of data so that the master controller can stop HASPRNG operation. Figure 19 describes the local controller implementation. The local controller has 5 input ports and 4 output ports. The two types of enable signals make the data transfer from the controller to the dual port RAM effective. The local enable signal comes from the HASPRNG module, and the global enable signal comes from the PowerPC. The local enable signal is used to activate the controller module. An address generator is instantiated in the controller. The address generator increments the address variable value by '4' at every clock until it reaches the last address in the dual port RAM. When it reaches the last address, the address variable value does not change. In order to activate the address generator again, the address variable value must be changed. The global enable signals initialize the address variable which is the first address in the dual port RAM, and makes the address generator active again. The global enable signal is also used to activate the HAPRNG module. The 32bit output data is a random number which is stored to the dual port RAM as shown Figure 20.



```
ENTITY controller IS
PORT(
clock : in std_logic;
reset : in std_logic;
en_1 : in std_logic;
en : in std_logic_vector(31 downto 0);
datain : in std_logic_vector(31 downto 0);
dataout: out std_logic_vector(31 downto 0);
addr: out std_logic_vector(16 downto 0);
we: out std_logic;
full: out std_logic);
END controller;
```

Figure 19. Local Controller



```
Entity mem_bank32768x32
port (addra: IN std_logic_vector(16 downto 0);
      addrb: IN std_logic_vector(16 downto 0);
      BEa : IN std_logic_vector(3 downto 0);
      BEb : IN std_logic_vector(3 downto 0);
      clka: IN std_logic;
      clkb: IN std_logic;
      dinb: IN std_logic_vector(31 downto 0);
      douta: OUT std_logic_vector(31 downto 0);
      ena: IN std_logic;
      enb: IN std_logic;
      web: IN std_logic);
end mem_bank32768x32;
```

Figure 20. Dual Port RAM Entity

The master controller activates and stops HASPRNG operation. This task is relevant to not merely the verification platform but also the programming interface environment. The programming interface environment will be discussed in chapter 5. The internal master controller states are characterized by 4 states in Figure 21. The master controller produces the enable modification signal. In the initial state, the modification signal is set to “0” which does not affect the module operation. The actual operation starts after getting the full flag signal from the local controller. Once the master controller takes the memory full flag signal from the local controller, the master controller sends the enable modification signal having a value “FFFFFFFF” through the output port. The global enable signal is modified by a XOR operation with the signal “FFFFFFFF”. The modified global enable signal stops HASPRNG operation. While the HASPRNG module pauses, the PowerPC starts to read the data from the dual port RAM and checks the data with the one from SPRNG. When it completes the check for the 32K random numbers, the PowerPC gives a signal to generate a latch signal. The latch signal prevents the master controller from going to the next state without a latch signal acknowledgement. The last state is the enable state. Once the master controller generates a latch signal, the master controller activates the HASPRNG module again. The 2, 3, and 4 states are performed iteratively. Figure 22 describes the master controller implementation.

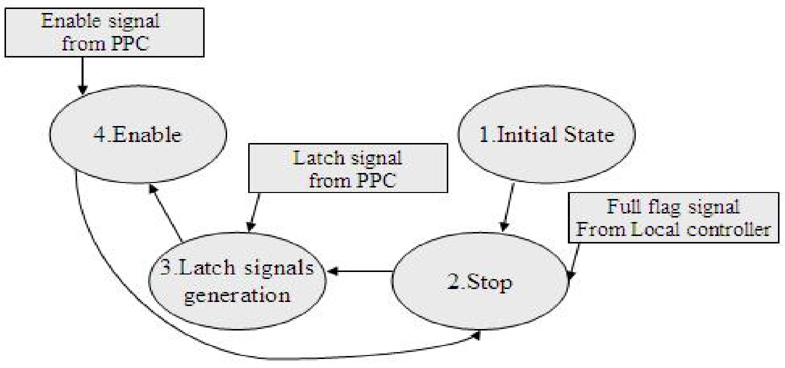
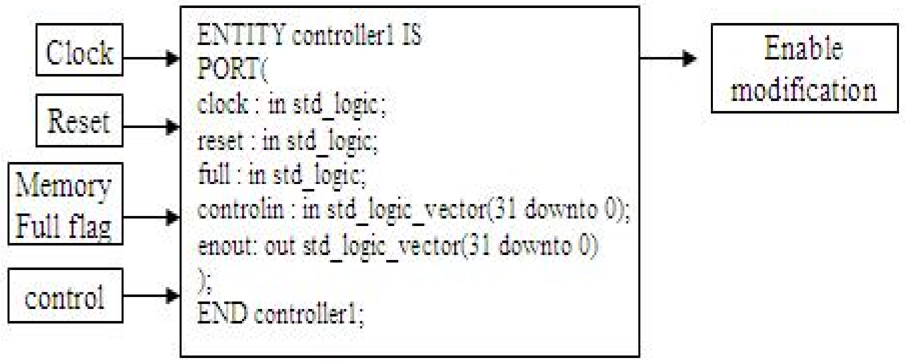


Figure 21. Verification Signals Flow



```
ENTITY controller1 IS
PORT(
clock : in std_logic;
reset : in std_logic;
full : in std_logic;
controlin : in std_logic_vector(31 downto 0);
enout: out std_logic_vector(31 downto 0)
);
END controller1;
```

Figure 22. Master Controller

The master controller has 4 input ports and 1 output port. When the controller receives the memory full flag signal from the local controller, it generates the enable modification signal value "FFFFFFFF". A HASPRNG enable signal is modified by XOR operation with the enable modification signal. When the enable modification signal is not '0', the HASPRNG module's operation pauses. When the random numbers check in the dual port RAM is done through the PowerPC, the PowerPC gives the signal to the controller through the control input port. When the controller takes the signal, the controller enables the HASPRNG module again by assigning the value '0' to the enable modification output.

4.3. Software Platform for Verification

The HASPRNG verification platform is written in C++. The architecture subsection describes how we compare HASPRNG data with SPRNG data. In order to compare HASPRNG data to SPRNG data, SPRNG was ported to the PowerPC platform. The implementation subsection discusses the porting of SPRNG to the PowerPC platform.

4.3.1. Architecture

The software platform employs C++ for the interface between HASPRNG and SPRNG. The source code uses the SPRNG library. Inside the code, the PowerPC reads the data by using a function from the SPRNG library and compares the data to the HASPRNG data in the dual port RAM. The 32bit data are checked using an XOR operation. If a data check fails, the source code gives an error message and the PowerPC terminates the verification process.

4.3.2. Implementation

SPRNG is ported to the PowerPC platform to allow faster verification. This section describes how we port SPRNG to the PowerPC platform for verification.

4.3.2.1. Retargeting SPRNG on XC2VP30 platform

SPRNG 4.0 was targeted to the PowerPC platform using two Makefiles which SPRNG provides. The first Makefile is used to choose a platform. For the retargeting, GENERIC was chosen for the PowerPC platform. The second Makefile performs compilation of the codes and generates a library file. Table 1 shows the generation rate from the SPRNG using the PowerPC platform in the XC2VP30.

TABLE 1. SPRNG 4.0 Generation Speed on PPC 405 Platform

	LFG	LCG	LCG64	CMRG	MLFG	PMLCG
Duration for 1MRNs	9 sec	18 sec	1300 sec	1600 sec	240 sec	20 sec
MRNs	0.1111	0.0555	0.0008	0.0006	0.0042	0.05
RNs/sec(Hz)	111 K	55 K	0.8 K	0.6 K	4 K	50 K

In the Makefile, some modifications were necessary. The archive (ar) command was changed to powerpc-eabi-ar, the ranlib command was changed to powerpc-eabi-ranlib, and the compiler (g++) was changed to powerpc-eabi-g++. SPRNG includes FORTRAN wrapper files. All the FORTRAN wrapper files were removed. The sprng.cpp file includes several functions related to timing which were disregarded since they were not necessary for the verification. Once the library file is obtained, the library file is accessed from the main source code. When a user tries to compile the main source code using the SPRNG library, the powerpc-eabi-gcc compiler should be linked with the option (-stdc++, -supc++) in order to access the C++ library. When the main source code is run, the random numbers from SPRNG are compared to HASPRNG. In order to get coefficient values from SPRNG, several functions are implemented additionally for SPRNG. For example, in order to get the seed and the prime values from the 48 bit LCG, the get_seed_48_0() and get_seed_48_1(), and get_prime_48() functions are instantiated inside SPRNG. Figure 23 shows the functions additions to SPRNG for the verification and programming interface for HASPRNG. These added functions are used for obtaining the initial coefficients from SPRNG directly. These coefficients are fed into the HASPRNG during initialization.

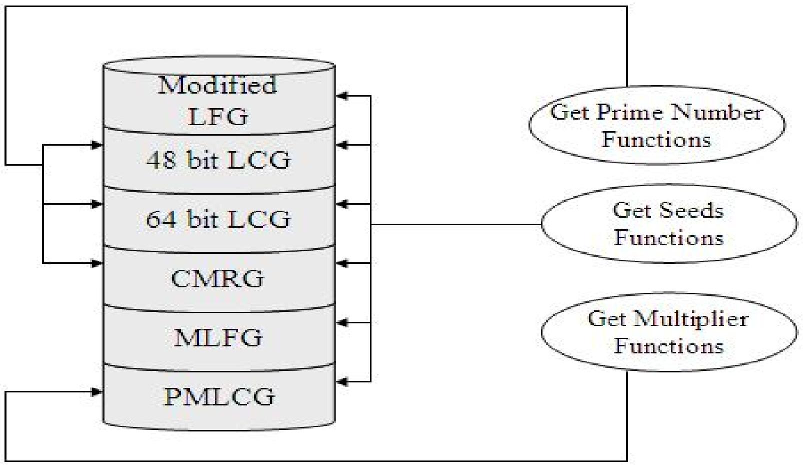


Figure 23. Added Functions to SPRNG

4.3.2.2. Source Code Implementation

The Xilinx Platform Studio (XPS) provides the interface environment between software and hardware so that the PowerPC can access data from the FPGA directly. XPS supports software interface registers and a memory address range option in order to access data in the FPGA.

In the verification code, the random numbers are checked one by one. The source code implementation is described as shown in Figure 24. When the dual port RAM is full of data, the HASPRNG module operation pauses. The verification code then compares the results from HASPRNG and SPRNG. The for-statement is used to read and compare 32K random numbers. The verification code reads a random number data from SPRNG using a function call, and compares the data with the HASPRNG one. The verification code increases the address variable by 4 to read the next random number of HASPRNG. The compare operation is done by XOR bit operation. When an XOR operation result has any value except ‘0’, the verification is terminated, escaping the while loop and showing an error message.

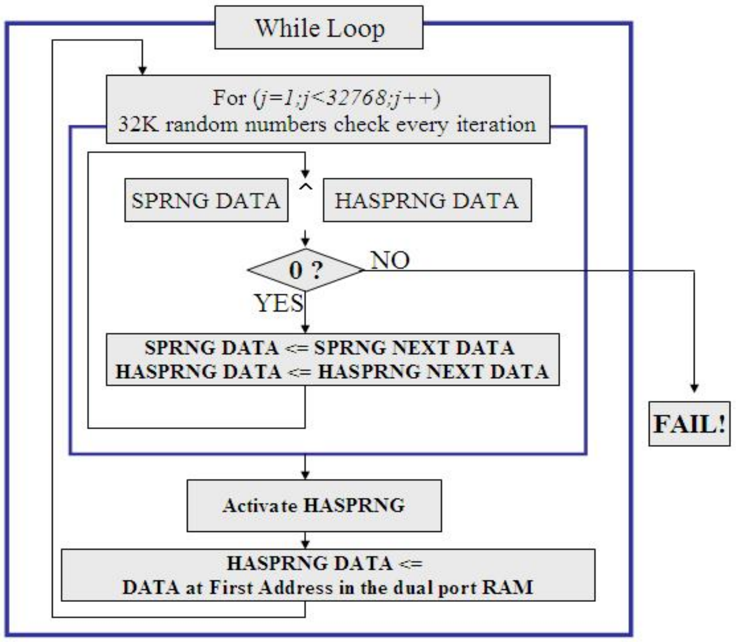

While Loop
For (j=1;j<32768;j++)
32K random numbers check every iteration
SPRNG DATA
^
HASPRNG DATA
0 ?
NO
YES
SPRNG DATA <= SPRNG NEXT DATA
HASPRNG DATA <= HASPRNG NEXT DATA
FAIL!
Activate HASPRNG
HASPRNG DATA <=
DATA at First Address in the dual port RAM

Figure 24. Source Code Implementation

Once the current 32K random numbers in the dual port RAM are checked, the verification code activates the hardware again, and the next 32K random numbers are inserted in the dual port RAM.

4.4. Verification Results

All the HASPRNG generators are verified using the verification platform above. The verification speed depends on the SPRNG generation speed, because HASPRNG is much faster than the SPRNG generation on the PowerPC platform. All the modules are verified with over 1 million numbers, with most parameter sets given by SPRNG and several seeds.

4.5. Summary

The verification platform employs a bitwise check for a random number verification of HASPRNG. SPRNG is ported to the PowerPC in the XC2VP30 FPGA in order to verify HASPRNG effectively. Two controllers are employed to support seamless data transfer from HASPRNG for the verification platform. On the verification platform, all the generators in HAPRNG are verified with over 1 million consecutive numbers.

Chapter 5. Programming Interface

Fortunately, the verification platform provides a user-friendly programming interface. SPRNG generates a random number on demand with a function call. In consequence, HASPRNG can provide a software implementation function which generates a random number. By providing a similar programming interface, integration with applications becomes much easier. The programming interface is primarily a software implementation since the hardware platform for verification already allows the PowerPC to read random number data. This chapter illustrates how to implement the user-friendly programming interface.

Many applications developers may wish to employ the faster random number generation offered by HASPRNG, but they may not have the training or the inclination to do low-level VHDL programming to interface to HASPRNG. A design goal for HASPRNG is to provide a similar programming interface as with SPRNG in order to hide details of logic design and FPGA interfacing. In doing so, a much broader community of users can employ HASPRNG and hardware acceleration, but with virtually no additional development challenges.

5.1. Overview

The programming interface allows a user to link with HASPRNG exactly the same way as with SPRNG. SPRNG uses several functions to perform random number generation. For example, SPRNG provides the init_sprng() function to initialize a random number generator, isprng() to generate a random number, and print_sprng() to show the random number generator status. The random number generator status is characterized by the generator type, the initial seed value, and the parameter set. In order to provide a user with the same environment as SPRNG, some of the SPRNG functions are employed directly. For example, in the programming interface, the init_sprng() function from SPRNG is used in HASPRNG directly to extract the initial coefficient values, such as the initial seeds, and the prime numbers. It is advantageous to obtain the initial coefficient values directly from the SPRNG initialization function, since the initialization function performs its operation just one time to get the desirable coefficient values. All the initial coefficient values are determined when the initialization process in the function is complete. Since SPRNG is ported to the PowerPC platform, it is possible to use the SPRNG functions directly in the source code.

The programming interface provides faster access to a random number than SPRNG. SPRNG waits until the generation execution is complete and reads the data. In the programming interface for HASPRNG, the 32K generation execution is done by the FPGA at once, since the dual port RAM size in the FPGA can accommodate 32K random numbers. Once 32K random numbers are stored in the dual port RAM, the data is read by a function. Hence, we can expect performance improvement for the random number generation.

5.2 Implementation

Consider the programming interface as applied to the 48 bit LCG. HAPRNG stores data to the dual port RAM in the FPGA, and the PowerPC reads the data one by one from the dual port RAM. The functions needed to be considered are the initialization function (init_sprng()), and the generation function (isprng()). For the implementation for the programming interface, the additional class named HASPRNG is declared. The following description shows the class for the HASPRNG.

```
class Hasprng
{ public: int addr;  int init_addr;  int ihasprng();  int init_hasprng(); };
```

Two member variables and two member functions are necessary for the user-interface environment. The member variables are accessed by the member functions. The addr member variable changes value over the course of the member function operations. The init_addr member variable value does not change. It keeps the first address of the dual port RAM.

The initialization function in the HASPRNG class performs two tasks. The first one is to give the initial coefficient values, such as the initial seed, the prime number, and multiplier coefficient values for the 48 bit LCG, to HASPRNG for its operation. The other task is to perform the HASPRNG operation one time so that it can store the data into the dual port RAM. The initialization function is described as shown in Figure 25. The HASPRNG initialization function (init_hasprng()) accesses the member variables related to address (addr, and init_addr). The source code for the initialization function starts with the declaration. The initialization function is one of the member functions in the HASPRNG class. The declaration has to use "::" notation to indicate the function is one of the member functions of the class. The function takes three arguments, which allows a user to access it the same way as the SPRNG initialization function. For example, the init_sprng() function in SPRNG takes three arguments for the initialization.

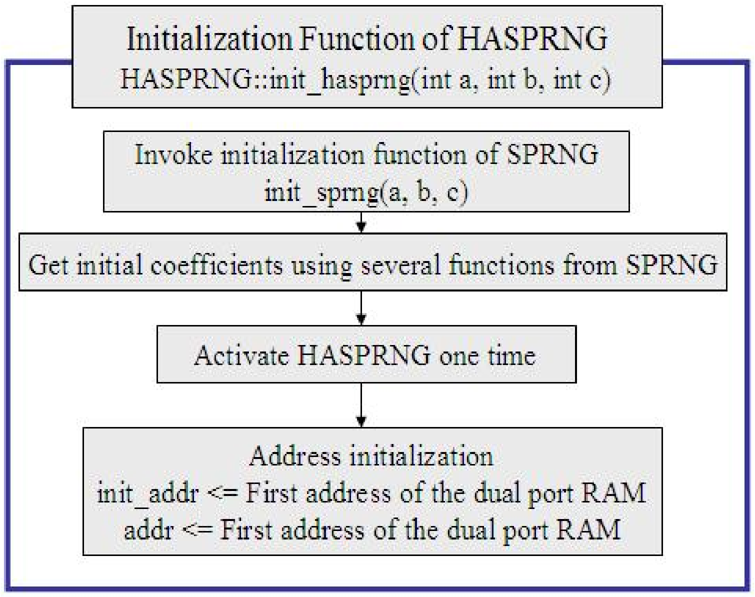

Initialization Function of HASPRNG
HASPRNG::init_hasprng(int a, int b, int c)
Invoke initialization function of SPRNG
init_sprng(a, b, c)
Get initial coefficients using several functions from SPRNG
Activate HASPRNG one time
Address initialization
init_addr <= First address of the dual port RAM
addr <= First address of the dual port RAM

Figure 25. Initialization Function of HASPRNG

Once the HASPRNG initialization function obtains three arguments, those values are transferred to the SPRNG initialization function. The SPRNG initialization function is called inside the HASPRNG initialization function. The SPRNG initialization function produces the initial coefficient values, such as the initial seed, the prime number, and the multiplier coefficient value. Several functions, described in Chapter 4, are able to take the initial coefficient values. Those values are fed into a HASPRNG module's inputs by some functions calls. When the HASPRNG module gets its initial variables, the controller gives an enable signal to the module to fill up the 128K Byte dual port RAM with random numbers. The address member variables are also initialized to the first address in the dual port RAM. Once the initialization function process is complete, the generation function reads the data from the dual port RAM.

The main task of the generation function is to read data from the dual port RAM in order to return a random number by a function call. Figure 26 describes the generation function. When the generation function accesses the last data of the dual port RAM, it activates a HASPRNG module so that the dual port RAM can contains the next 128K Byte data.

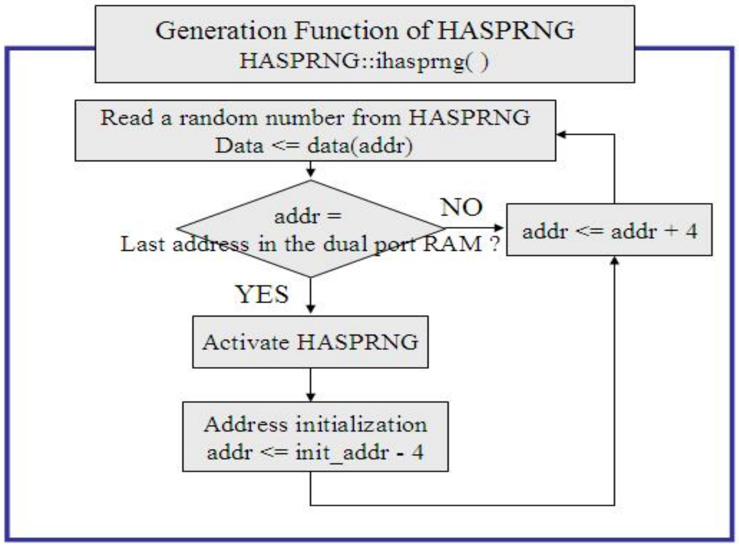


Figure 26. Generation Function of HASPRNG

The next function call reads the new data from the dual port RAM. The address increments by 4 in order to indicate the next word size (4 Bytes) data, since a random number size is 4 Byte.

Those two functions can provide a programming interface so that the user can use HASPRNG library the same way as SPRNG. The executable file for the programming interface code is stored at some portions of the SDRAM, and the XMD tool executes the file.

The programming interface was tested with over 1 million random numbers by comparing the results with those from SPRNG on each generation function call. The random number generation rate for the 48 bit LCG is much slower than the FPGA generation rate. The 48 bit LCG generates 100 million random numbers per second in the FPGA with no consideration of hardware overhead. With the programming interface, it takes 5 seconds to generate 1 million random numbers. When you see the 48 bit LCG generation speed results in Chapter 4, it indicates that 18 seconds are required for generating 1 million random numbers for the 48 bit LCG. That means it introduces 3.6 times performance improvement on the PowerPC platform. Even though I do not get desirable results for the performance for the programming interface code on the XC2VP30 platform, we can expect the better performance results on a real application platform, since most time it takes on the interface is caused by the read and write operations, and the function calls from the PowerPC.

The XUP development board does not provide a high-speed interface to the FPGA. The high-speed interface is provided in several systems such as the Cray XD1, SGI RASC, or SRC MAP station. For example, the 32K random numbers from HASPRNG are stored in the dual port RAM in FPGA, and the random numbers are read by the PowerPC on the XUP board. In a real application system, the 32K random numbers in the FPGA will be transferred to the main memory near the microprocessor. After transferring the data, the microprocessor reads the data in stead of generating random numbers. Assuming the transferring cost is small because of the support of the high-speed interface, the time it takes for generating a random number is considerably faster because the microprocessor does not need to generate a random number and it reads the data quickly. We seek to port HASPRNG to such systems to provide high performance random number generation to users of such systems.

5.3. Summary

The programming interface allows a user to use HASPRNG the same way as SPRNG. The performance using the programming interface is not desirable on the XUP board. The XUP is a development board, so it does not provide the high-speed interface. However, if it is applied to a real application system providing the high-speed interface, we can expect much better performance.

Chapter 6. Results

HASPRNG shows identical results to SPRNG. Also, HASPRNG introduces significant performance improvements compared to SPRNG. This chapter shows the results in detail in terms of functionality, hardware usage, and performance.

6.1. Functionality Verification Results

The functionality verification is the most basic and important task for the project. Even though we get a high performance result, it is meaningless unless a HASPRNG module produces identical results with SPRNG, otherwise a user cannot use the HASPRNG module for their SPRNG application.

All HASPRNG generators and parameters produce identical data to SPRNG. Every generator in the HAPRNG is verified with over 1 million random numbers using a bitwise XOR operation.

6.2. Performance Results

Performance motivates all of this research. Next, we compare SPRNG and HASPRNG performance.

6.2.1. SPRNG Performance

The SPRNG website provides performance reports for several older machines [2]. However, it does not show the results for a modern Pentium class machine. The SPRNG library performance was evaluated on a 2.8GHz Pentium 4 processor for updated performance comparisons. The SPRNG performance was tested with two different versions (SPRNG 2.0b, and SPRNG 4.0). Table 2 shows the SPRNG performance summary according to different machine architectures.

Table 2. SPRNG Performance

MRNs	Modified LFG	48 bit LCG	64 bit LCG	CMRG	MLFG	PMLCG
Cray T3E	5.5	14.5	16.6	8.6	5.6	7.5
HP/Convex	6.7	7.7	10.0	5.6	5.3	0.7
IBM SP 2	4.0	4.4	5.6	1.9	4.2	2.2
SGI Cray Origin 2000	6.1	10.3	11.5	4.6	8.3	2.5
SGI Power Challenge	5.8	10.2	11.5	4.7	8.3	2.4
Pentium IV 2.8GHz SPRNG Ver 2.0b	18.3	17.9	5.1	2.9	10.0	7.3
Pentium IV 2.8GHz SPRNG Ver 4.0	19.3	20.3	6.6	3.2	10.4	7.3
PowerPC in XC2VP30 [XUP] –SPRNG 4.0	0.1111	0.0555	0.0008	0.0006	0.0042	0.05

The numbers in the table represent millions of random numbers generated per second. When you observe the table, the Pentium 4 processor represents the best performance in most cases. In the 64 bit LCG, the Cray machine represents the best result in the table. The Cray machine also represents the best result in the CMRG, since the CMRG employs the 64 bit LCG as a part of the generator. The PMLCG performance is similar between the Pentium 4 and the Cray machine. The generation speed for the PowerPC on the XUP board is considerably slower compared to other real application systems, because the PowerPC is an embedded processor with low clock speed and minimal memory. The table shows that the SPRNG 4.0 design is slightly better optimized for performance than the previous version.

6.2.2. HASPRNG Performance on XC2VP30

For HASPRNG performance, hardware usage and maximum clock frequency are described separately. Maximum frequencies after place and route and the generation speeds for each of the generators are described.

6.2.2.1. Hardware Usage

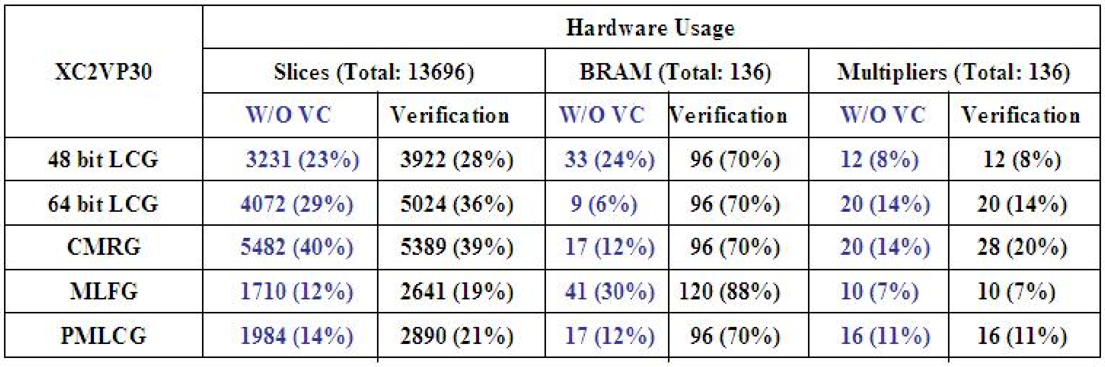
The hardware usage is an interesting area for the results, since FPGA hardware resources are limited. Even though performance is considerably faster, it is meaningless unless it fits into a FPGA. Table 3 shows hardware resource usage. This usage is for HASPRNG without the verification platform and with the verification platform having a large buffer.

Under the verification platform, all the generators' hardware resources include the dual port RAM whose size is 128KByte. The slices and the BRAM size can be changed according to the dual port RAM size. The number in a parenthesis represents the usage percentage of the resources in the XC2VP30 FPGA. All the multipliers in the generators are the built in 18 x 18 multipliers.

6.2.2.2. Maximum Clock Frequency

XPS reports the maximum allowable clock cycle for a module after the place and route process. Table 4 shows maximum allowable clock rates according to the generators.

Table3. HASPRNG Hardware Usage on XC2VP30



XC2VP30	Hardware Usage					
	Slices (Total: 13696)		BRAM (Total: 136)		Multipliers (Total: 136)	
	W/O VC	Verification	W/O VC	Verification	W/O VC	Verification
48 bit LCG	3231 (23%)	3922 (28%)	33 (24%)	96 (70%)	12 (8%)	12 (8%)
64 bit LCG	4072 (29%)	5024 (36%)	9 (6%)	96 (70%)	20 (14%)	20 (14%)
CMRG	5482 (40%)	5389 (39%)	17 (12%)	96 (70%)	20 (14%)	28 (20%)
MLFG	1710 (12%)	2641 (19%)	41 (30%)	120 (88%)	10 (7%)	10 (7%)
PMLCG	1984 (14%)	2890 (21%)	17 (12%)	96 (70%)	16 (11%)	16 (11%)

Table 4. Maximum Allowable Clock Frequency after PAR

	48 bit LCG	64 bit LCG	CMRG	PMLCG	MLFG	Modified LFG (O/ O)	Modified LFG (O/E)
XV2VP30	103 MHz	101 MHz	101 MHz	78 MHz	68 MHz	101 MHz	105 MHz

Most of the generators meet the timing constraints when the On-chip Peripheral Bus (OPB) is set to 100 MHz. The MLFG and PMLCG do not meet the timing constraint of 100 MHz. Because of that, the OPB clock for the MLFG is set to 66.7MHz. Even though the PMLCG does not meet the timing constraint of 100MHz, it is ported to the 100MHz OPB environment. The CMRG and PMLCG use the Synplicity synthesis tool to get netlists in order to improve the allowable maximum frequency. The rest of the generators use the XST tool to get netlists.

The Modified LFG, MLFG, and 48 bit and 64 bit LCGs generate a random number every clock cycle. The CMRG and PMLCG generate a random number every other clock cycle. The maximum generation rate can be obtained according to the OPB setting. All the generators except the MLFG work on a 100MHz OPB, and the MLFG works on 66.7 MHz OPB. In the case of the CMRG and PMLCG, the maximum generation rate is divided by 2 from their OPB clock frequency. Table 5 shows the maximum generation rate according to the generator types. The MRNs represents 1 million random number generation per second.

6.2.3. Comparison of Performances

Once the functionality verification is complete, the next step is to evaluate the performance for the generators. Table 6 shows comparison results. HASPRNG should represent a considerable performance improvement compared to the SPRNG in order to attract a user to the HASPRNG library. Fortunately, the HASPRNG introduces 4-70 times performance improvement compared to SPRNG.

Table 5. Random Number Generation Speeds of HASPRNG

	48 bit LCG	64 bit LCG	CMRG	PMLCG	MLFG	Modified LFG(O/ E)	Modified LFG(O/E)
XV2VP3 0	100 MRNs	100 MRNs	50 MRNs	50 MRNs	66.7 MRNs	100 MRNs	100 MRNs

Table 6. Performance Comparisons of HASPRNG with SPRNG

(MRNs: million numbers per a second)

	Modfied LFG	48 bit LCG	64 bit LCG	CMRG	MLFG	PMLCG
Cray T3E	5.5	14.5	16.6	8.6	5.6	7.5
HP/Convex	6.7	7.7	10.0	5.6	5.3	0.7
IBM SP 2	4.0	4.4	5.6	1.9	4.2	2.2
SGI Cray Origin 2000	6.1	10.3	11.5	4.6	8.3	2.5
SGI Power Challenge	5.8	10.2	11.5	4.7	8.3	2.4
Pentium IV 2.8GHz SPRNG Ver 2.0b	18.3	17.9	5.1	2.9	10.0	7.3
Pentium IV 2.8GHz SPRNG Ver 4.0	19.3	20.3	6.6	3.2	10.4	7.3
XC2VP30 FPGA(XUP)	100	100	100	50	66.7	50
XUP FPGA Speed Up (Min)	5.2X	4.9X	6.0X	5.8X	6.4X	6.7X
XUP FPGA Speed Up (Max)	25.0X	22.7X	19.6X	26.3X	11.9X	71.4X
XUP FPGA Speedup vs. P 4	5.2X	4.9X	15.2X	15.6X	6.4X	6.8X

HASPRNG shows 4-15 times performance improvement over the Pentium 4 per copy of the random number generators. In general, the HASPRNG shows a 5 times performance improvement compared to the SPRNG with no hardware overhead consideration. Even though these improvement factors do not consider hardware overhead, sufficient performance improvement is expected when HASPRNG is ported to a real application system supporting high speed interface.

6.3. Expected Results on Virtex 4

The Synplicity synthesis tool generates the netlists for all the generators for the XC4VFX100 platform. The multipliers for the netlists are changed to the LUT multipliers, since the XC4VFX100 FPGA does not have an 18×18 multiplier. Even though the FPGA has DSP multipliers, their pipelined stage is different than the 18×18 multipliers. The Xilinx ISE 8.2 is employed to get the allowable maximum frequency results for the generators. Table 7 describes the maximum delay, and the maximum frequency results for the XC4VFX100 platform.

Table 7. XC4VFX100 Maximum Clock Frequency after PAR

	48 bit LCG	64 bit LCG	CMRG	PMLCG	MLFG
XC4VFX100	4.168ns	4.556ns	10.743ns	12.682ns	12.496ns
XC4VFX100	239.923MHz	219.491MHz	93.084MHz	78.852MHz	80.026MHz

In the case of the 48 bit LCG, it reports 240 MHz for the maximum allowable frequency. When you observe the 48 bit LCG architecture, the 48 bit LCG uses deeply pipelined multipliers, and does not have long combinational logic delays between registers. However, the PMLCG reports the worst result out of the generators, since it employs heavy combinational logic between registers to save clock cycles as discussed in Chapter 3.

6.4. Summary

HAPRNG produces identical results with SPRNG. HASPRNG represents 4-15 performance improvement over SPRNG in 2.8 GHz Pentium 4. Each HASPRNG generator consumes 12-40% of slices, 6-30% of BRAMs, and 7-14% of multipliers in the XC2VP30 FPGA. Considering the verification platform, the percentage increases slightly for slices and multipliers, and considerably for BRAMs.

`Chapter 7. Conclusions and Future Work

Discussions subsection describes the challenges for the implementation, and debugging process with HASPRNG. The conclusions, future work and contributions are discussed in this chapter as well.

7.1. Discussions

The five generators in HASPRNG are designed to maximize the performance. The main issues from the five generator implementations come from data hazards. The two LCGs, CMRG, and PMLCG use the previous random number to generate a random number. The MLFG also uses a previous random number represented by a lag factor. All the five random number generators perform multiplications to generate a random number. The data sizes are large for the five generators. If the input data for a multiplier is large, the multiplier cannot perform its operation within one clock cycle, which causes a data hazard.

In the case of the two LCGs, the original equation is transformed to another one which is fit for hardware implementation. The new transformed equation allows the LCGs to use deeply pipelined multipliers to overcome the data hazard. The two LCGs generate a random number every clock cycle. The CMRG and PMLCG could not overcome the data hazard, since it is tremendously hard to predict a future random number. In other words, it is extremely hard to find out the recursion relations inside their characteristic equations. By using a forwarding scheme for CMRG and PMLCG, we save a clock cycle to generate each random number. The generators generate a random number every other clock cycle. The MLFG uses a 2 staged pipelined multiplier, and generates a random number every clock cycle. The MLFG is limited to 66.7 MHz because of a 2 staged pipelined multiplier. We need to investigate the mathematical expressions since a slight tweak to the mathematical expressions can make the hardware implementation much better in terms of hardware resources and performance.

Many unexpected events occurred during the development. For example, the PMLCG design employs 4 multipliers, which have different input data size. For example, one port of two inputs has 30 bit data size, and the other port has 31 bit data size. When we employed a multiplier having different input data size, the output data was not correct. After fixing all 4 multipliers to the multipliers having 32bit × 32bit inputs, the bug was fixed. The 48 bit LCG design uses a big state machine. When I used several state machines inside a state machine, the module did not work properly, even though it worked in pre-synthesis simulation. After unfolding those state machines to one state machine, the bug was fixed. It is desirable to make synthesizable VHDL designs so that a synthesis tool can understand what a designer intends to do.

I recommend using a verified IP core to design a VHDL module. In the PMLCG implementation, I employed a clock divider to activate the local controller to transfer data. The clock divider was described by some VHDL codes. The controller did not transfer a correct data. The bug was fixed after removing the clock divider and employing the original clock with an extra signal inside the local controller module.

7.2. Conclusions

HASPRNG includes the full set of SPRNG. All the generators in HASPRNG produce identical results to SPRNG, and are verified with substantial quantity of random numbers. Also, HASPRNG provides a user-friendly programming interface by employing one local controller and one master controller. HASPRNG generation in FPGA shows considerable performance improvement compared to SPRNG. Applying the programming interface on XUP does not produce desirable performance results due to heavy hardware overhead caused by function calls and reading data from the dual port RAM. When HASPRNG is ported to a system such as the Cray XD-1, not a development system (XUP), we can expect much better performance due to more efficient communications. We believe that the performance improvement by HASPRNG will contribute to computational science.

7.3. Future Work

The most important future work is to port HASPRNG to the Cray XD-1 machine. In order to complete the future work, getting information on the Cray XD-1 interface for the FPGA is necessary. Even though a programming interface environment is realized on XUP, a better approach should be investigated in order to reduce hardware overhead costs.

7.4. Contributions

HASPRNG has realized a full set of SPRNG. HASPRNG produces identical results to SPRNG, and provides the programming interface. Even though a user may not have hardware knowledge, the user can use HASPRNG the same as SPRNG. In other words, HASPRNG provides a user the same results and usage as SPRNG for integer random number generation. HASPRNG will be ported to Cray XD-1 machine soon. After porting to the real application platform, a HASPRNG will provide not merely the good quality of random number generators of SPRNG but also a performance improvement over SPRNG. Many research areas employ MC simulations which use huge quantities of random numbers. In consequence, MC simulations require random number generators having good quality and high performance. Based on HASPRNG characteristic having good quality, high performance, and the same usage as SPRNG, we expect that HASPRNG will be widely used for MC simulations applications in the future.

LIST OF REFERENCES

LIST OF REFERENCES

[1] M. Mascagni, D. Ceperley, and A. Srinivasan, "Algorithm 806: SPRNG: A scalable library for pseudorandom number generation," *ACM Transactions on Mathematical Software*, vol. 26, no. 3, pp. 436-461, September 2000.

[2] Scalable Parallel pseudo Random Number Generators library, SPRNG. [Online]. Available: http://sprng.fsu.edu.

[3] Y. Bi, G. D. Peterson, G. L. Warren, and R. J. Harrison, "Hardware acceleration of parallel lagged Fibonacci pseudo random number generation," in *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms*, CSREA, 2006.

[4] Y. Bi, "A reconfigurable supercomputing library for accelerated parallel lagged-Fibonacci pseudorandom number generation", Master of Science Thesis, Department of Electrical and Computer Engineering, University of Tennessee, December 2006.

[5] J. Lee, G. D. Peterson, and R. J. Harrison, "Hardware accelerated scalable parallel random number generators", in *Proc. Reconfigurable Systems Summer Institute.* RSSI, 2007

[6] National Center for Supercomputing Applications, NCSA. [Online]. Available: http://access.ncsa.uiuc.edu/CoverStories/RandomNumbers.

[7] C. Khor, "XUP-V2P Tutorial", Master of Science Project Report, Department of Electrical and Computer Engineering, University of Tennessee, November 2005.

[8] J. L. Tripp, A. A. Hanson, M. Gokhale, and H. Mortveit, "Partitioning hardware and software for reconfigurable supercomputing applications: A case study", in *Proc. Supercomputing*, SC, 2005.

[9] I. Vattulainen, K. Kankaala, J. Saarinen, and T. Ala-nissila, "A comparative study of some pseudorandom number generators", *Computer Physics Communications*, vol. 86, no. 3, pp. 209-226, May 1995.

[10] J. M. McCoolum, J. M. Lancaster, D. W. Bouldin, and G. D. Peterson, "Hardware acceleration of pseudo-random number generation for simulation applications", in *Proc. 35th IEEE Southeastern Symposium on System Theory,* SSST, 2003.

[11] P.P. Chu and R.E. Jones, “Design techniques of FPGA based random number generator (extended abstract)”, in *Proc. Military and Aerospace Applications of Programming Devices and Techniques,* MAPLD, 1999.

[12] A. P. Paplinski and N. Bhattacharjee, "Hardware implementation of the Lehmer random number generator", *IEE Proc.-Computer and Digital Techniques,* vol. 143, no. 1, pp. 93-95, January 1996.

[13] Embedded Systems Tools Reference Manual, EDK 8.2, June 2006 [Online]. Available: http://www.xilinx.com/ise/embedded/edk_docs.htm.

[14] Xilinx University Program Virtex-II Pro Development System, March 2005 [Online]. Available: http://www.digilentinc.com/Data/Products/XUPV2P/XUPV2P_User_Guide.pdf.

[15] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W.Luk and P. Cheung “Reconfigurable computing: architectures and design methods”*, IEE Proc.-Computer and Digital Techniques*, vol. 152, no. 2, pp. 193-205, March 2005.

[16] A. Srinivasan, M. Mascagni, and D. Ceperley, “Testing parallel random number generators”, vol. 29, no. 1, pp. 69-94, January 2003.

[17] G. Cenacchi and A. De Matteis, “Pseudo-random numbers for comparative Monte Carlo calculations”, *Numerische Mathematik,* vol. 16, no. 1, pp. 11-15, February 1970.

[18] M. Mascagni and A. Srinivasan, “Parameterizing parallel multiplicative lagged-Fibonacci generators”, *Parallel Computing,* vol. 30, no. 7, pp. 899-916, July 2004.

[19] T. Warnock, “Random-number generators”, *Los Alamos Science Special Issue,* 1987.

[20] Y. Bi, G. D. Peterson, G. L. Warren, and R. J. Harrison, “A reconfigurable supercomputing library for accelerated parallel lagged-Fibonacci pseudorandom number generation”, *Supercomputing Conference,* SC, 2006.

[21] Quasi random number, July 2007, [Online]. Available: http://en.wikipedia.org/wiki/Quasi_random_number.

[22] P. D. Coddington and S. Ko, “Techniques for empirical testing of parallel random number generators”, in *Proc. Supercomputing,* SC, 1998.

[23] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, “High-performance reconfigurable computing”, *IEEE-Computer magazine,* vol. 40, no. 3, March 2007.

[24] M. C. Smith, S. L. Drager, Lt. L. Pochet, and G. D. Peterson, "High performance reconfigurable computing systems." in *Proc. IEEE Midwest Symposium on Circuits and Systems,* MWSCAS, August 2001.

[25] M. C. Smith and G. D. Peterson, "Parallel application performance on shared high performance reconfigurable computing resources", *Performance Evaluation,* vol. 60, no. 1-4, pp. 107-125, May 2005.

[26] Xilinx University Program, March 2006 [Online]. Available:
http://www.eece.unm.edu/xup/microblazeppc.htm.

[28] D. B. Thomas and W. Luk, "High quality uniform random number generation through LUT optimised linear recurrences", in *Proc. IEEE International Conference on Field-Programmable Technology,* ICFPT, 2005.

[29] W. Cui, C. Li, and X. Sun, "FPGA implementation of universal random number generator", in *Proc, the 7th IEEE International Conference on Signal Processing,* ICSP, 2004.

[30] D. Kagaris and S. Tragoudas, "A Design for testability techniques for test pattern generation with LFSRs", in *Proc. IEEE VLSI Test Symposium*, VTS, 1994.

[31] Linear Feedback Shift Register, LFSR, July 2007 [Online]. Available :
http://en.wikipedia.org/wiki/Linear_feedback_shift_register.

Vita

JunKyu Lee was born in Seoul, South Korea on March 9, 1974. He attended to HanYang University and received a B.S. in Physics and a B.S. in Electrical and Computer Engineering in 2002. From 1995 to 1998, he served for the Korean Air Force. He worked for the LG electronics Inc. in 2002 and for the NARA Controls Inc. in 2003 and 2004. He joined the Electrical and Computer Engineering department in the University of Tennessee, Knoxville in August 2005.

JunKyu Lee is currently pursuing his doctorate degree in Electrical Engineering at the University of Tennessee, Knoxville.