Application-adaptive pseudo random number generators and binding selector

Suk Han Lee[1], Ha Young Jeong[2], and Yong Surk Lee[3]
School of Electrical and Electronics Engineering, Yonsei University
134 Shinchon-Dong, Seodaemoon-Gu, Seoul 120-749, Korea
Tel: +82-2-2123-2872, Fax: +82-2-2123-4584
E-mail: {[1]shlee,[2]hyjeong}@dubiki.yonsei.ac.kr, [3]yonglee@yonsei.ac.kr

Abstract: In this paper, we propose hardware based random number generators according to an equation that uses area, speed, randomness level, and parallelism – depending on how many numbers are generated at a time. In addition, we propose a selector which can choose a suitable random generator among the others in the specific application using C language. Each random number generator is verified and synthesized. For each case, the instruction to delay random generation is compared with general software implementation. As a result, in keeping a rational trade-off line, the selector chose a suitable random generator which is about four times faster than software random number generation with a 2.25% area increase on average.

Keywords—**random number generator, ASIP, hardware implementation, reconfigurable hardware**

1. Introduction

Today, many applications require random numbers[1]. For example, to generate Additive Gaussian White Noise, RN16 on RFID tag[2], wireless communication encryption, random replacement policy in a cache controller[3], MMORPG(Massive Multiplayer Online Role Playing Game)'s random generation, and so on, random number generation is essential. However, most random number generators are working on software levels that take many CPU cycles. In this paper, to accelerate random number generation, four random number generation algorithms are chosen and implemented with a hardware level. In addition, an adaptive concept is used for choosing a random number generator with a reasonable line to satisfy area, speed, randomness, and the parallelism relationship.

2. Adaptive Random number generator

2.1 A concept of adaptive random generator

For choosing a suitable random generator in this paper, the adaptive fast reconfigurable concept is used[4]. According to each applications, as shown Figure 1, the main factors(area, power, randomness, and parallelism) are changed, and the random number generator is chosen. The notations and the equation will be described in section 2. 3.

2.2 Random number generation algorithms

In this paper, four algorithms – the simplest model to the most complex model – are used for implementation. Each of them have brief characteristics, and the advantages and disadvantages will be described for each part. All algorithms except the simplest one, have been implemented with software before. In this paper, they are implemented using synthesizable verilog-HDL as hardware models.

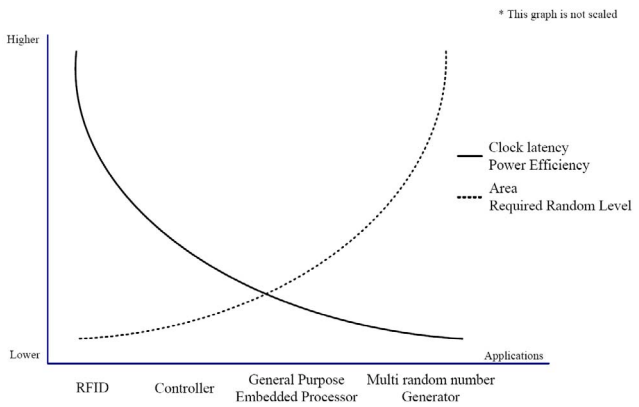

Figure 1. Characteristics of random number generation applications

2.2.1 The simplest model

A 4-bit Multiple-bit Leap Forward LFSR (Linear Feedback Shift Register) random generator is chosen, and its logical equation is shown below[5]. This one has already been implemented within a hardware model.

$$\begin{bmatrix} s0_next \\ s1_next \\ s2_next \\ s3_next \end{bmatrix} = A^{*} \begin{bmatrix} s0 \\ s1 \\ s2 \\ s3 \end{bmatrix}$$

A significant factor of the random number generation with LFSR is a transition matrix, and the sequence of the numbers is defined. It operates quickly and uses a small area. However, it generates a random number which has a low randomness level.

2.2.2 Simpler model

A random number generation algorithm using an adder is chosen[1].

$$x = x\hat{}rot(x,5)\hat{}rot(x,24)) + 0x37798849;$$

The function rot(variable,amount) refers to the rotation instruction. The 'variable' is shifted to 'amount'. In this architecture, it can be implemented with two 32bit XOR gates and an adder. It takes 5 cycles within software implementation [1]. It also operates quickly and uses a small area. However, it is not of good quality.

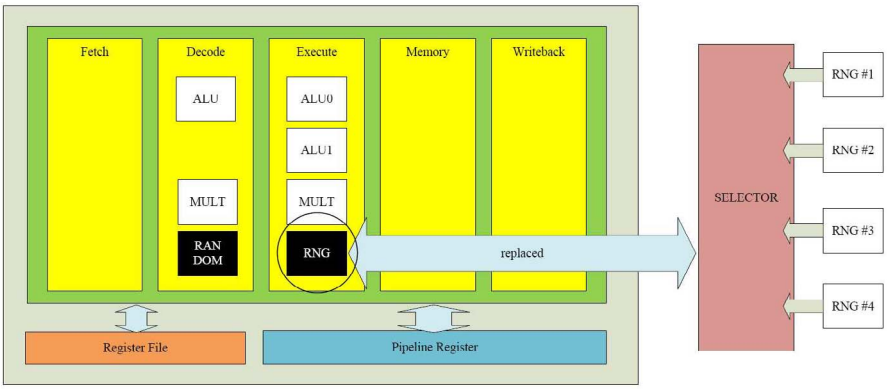

Figure 2. Unit generation and replacement in ASIP

2.2.3 General model

A random number generation algorithm that uses a counter and an adder is selected[1]. The parameters are (L,R,A)=(8,8,0x49A8D5B3). 'rot(x,L or R)' function main-

$$x = k++;$$
$$x = (x\hat{}rot(x, L)\hat{}rot(x, R)) + A;$$
$$x = (x\hat{}rot(x, L)\hat{}rot(x, R)) + A;$$
$$x = (x\hat{}rot(x, L)\hat{}rot(x, R)) + A;$$
$$x = (x\hat{}rot(x, L)\hat{}rot(x, R));$$
$$x = (x\hat{}rot(x, L)\hat{}rot(x, R));$$

tains the meaning as in the above model. In addition, this one takes 23 cycles upon software implementation [1]. This generator can make higher quality random numbers, but it takes a larger area.

2.2.4 Complex model

$x = k++; y = 0; z = 0; w = 0;$
$for(j = 0; j < B; j+= 4)\{$
$x\hat{}= rot(y + z + w, L)\hat{}A;$
$y\hat{}= rot(z + w + x, L)\hat{}A;$
$z\hat{}= rot(w + x + y, L)\hat{}A;$
$w\hat{}= rot(x + y + z, L)\hat{}A;$
$\}$
$for(j = 0;;)\{$
$if(++j > C)$
$break;$
$x\hat{}= rot(y + z + w, L);$
$if(++j > C)$
$break;$
$y\hat{}= rot(z + w + x, L);$
$if(++j > C)$
$break;$
$w\hat{}= rot(w + x + y, L);$
$if(++j > C)$
$break;$
$z\hat{}= rot(x + y + z, L);$
$\}$

A 4-stage generator is selected [1]. The parameters are (L,A,B,C)=(8,0x95A55AE9,12,3). This algorithm hardware implementation is much more complex than the others. It takes 18 cycles upon software implementation, and it needs 5 cycles on hardware implementation. Their randomness level is higher[1]. It takes a much larger area to have this random generator than the others, but it can make four random numbers simultaneously.

2.3 Proposed binding selection algorithm

Table 1 shows the variables mentioned in this paper. Each factor affects the final result directly, and an equation is derived.

$$CostFactor = k \times \frac{Randomness \times Parallelism \times Area}{Clockperiod}$$

The selector carefully calculates a cost factor with coefficient k. C program decides upon the most suitable generator among four different ones.

Table 1. Notations for given equation

Notation	Description
Randomness	random level (normalized)
Clock_period	clock period of machine (normalized)
Area	required area (normalized)
Parallelism	the number of random numbers at a time

2.4 Testing randomness

To verify the randomness of each random generator, a DIEHARD test suite[7] is used. The DIEHARD test suite is the defacto standard randomness test suite for random algorithm verification. Sixteen randomness tests in the DIEHARD test suit should be run on the given algorithm and the results represent a SUCCESS/FAILED. The p value in the DIEHARD test suite indicates a randomness level in each given test. When the p value is in the range of 0.001 through 0.998, it represents a SUCCESS in the given randomness test. The p value of over 0.999+ or a 0.000 value represents a

Table 2. Area, Clock Frequency, Randomness, and compare with software generation

Name	Area (NAND gate)	RN generation latency*	Parallelism	Randomness test Success	Randomness test Fail	RN generation clock latency with software*
LFSR driven	65k	1 cycle	1	1	15	N/A
Simpler model	66k	1 cycle	1	9	7	6 clks
General model	67k	6 cycles	1	15	1	23 clks
Complex model	70k	5 cycles	4	16	0	18 clks

*Pipeline penalties are not included

FAILED in the given test. So the randomness level of a pseudorandom generation algorithm can be presented as the number of the total SUCCESS or FAILED.

3. Implementation & Simulation

At first, a baseline RISC ASIP based on MIPS architecture is designed with LISA, which is developed by CoWare. LISA is an ADL(Architecture Driven Language), which can generate a processor with a coarse behavior model to synthesizable HDL code[6]. In addition, CoWare's processor designer provides SDK for a designed processor such as C-compiler, assembler, linker, debugger, and so on. Second, on the baseline ASIP, with two new instructions, as shown Table 3, are included to support random functions on an assembler level, so random instruction decode logic is generated. Third, we added an extension module to the baseline ASIP with LISA, and this generates a random number generation unit with behavior level. The interface of the designed random number generation unit is perfectly compatible with this automatically generated one. Thus, all designed random generators can be changed. Fourth, verilog code generation is performed. After generation, the selected pseudo random number generator is replaced with a chosen one, by the selector shown in Figure 2. Later, a replaced verilog HDL code is synthesized with a synopsys design compiler to observe the result. Finally, with Processor Debugger by CoWare, cycles are profiled. In addition, a generated number is verified.

Table 3. Added instruction set for random generator

Instruction	Description
rand <rd>	random number generation to register <rd>
srand <rd>	set the seed number from <rd>

4. Verification

The verification of the designed processor is done with lcc, which is a part of supported SDK by CoWare. Internal test vectors are used to verify the designed ASIP. All tests passed, completely.

In addition, pseudo random number generators which are designed with verilog HDL are verified with synopsys vcs verilog simulator. At first, the given formula is programmed with C, and compared with the result of the random number generators. All random number generators completely matched with the result of C.

The generated random number is verified with DIEHARD TEST SUITE. According to the result of DIEHARD TEST SUITE, LFSR random generator is not capable to verify randomness using the DIEHARD TEST because of the simpleness, so the the result of the LFSR model has no meaning. The LFSR model has 15 failed and 1 success, and the simpler model has 7 failed and 9 successes. The general model has 1 failed and 15 successes, and the complex model has zero failed.

5. Synthesis Result

The results of synthesis for each of the four cases with synopsys design compiler are shown in Table 2. The synthesized library is Dong-bu Hitec 0.18um library. Target clock latency is 10ns.

Synthesized result of the random generator shows its characteristic. LFSR driven shows the lowest performance with the smallest area. However, a complex model shows the highest performance with a larger area. Compared with the ASIP baseline, the increased area of LFSR, simpler algorithm, general one, complex one are 0.0%, 0.6%, 2.1%, and 6.2%, respectively. In addition, the increased speed compared with software implementation for the each algorithm is N/A, 6 times, 3.83 times, and 3.6 times, respectively.

6. Conclusion

In this paper, four random generators are implemented with hardware and a suitable random number generator for specific applications is chosen by a proposed selector. The proposed selector is based on a proposed equation. The selector chose a suitable random number generator among the set of random number generators before the synthesis level. As shown in the results, with a little increased area, an application which needs random numbers is accelerated with the selected random generator, which is suitable for each application, directly. As a result, the selector chose a random number generator with a reasonable line.

7. Acknowledgment

This work was supported by ETRI (Electronics and Telecommunications Research Institute) from the Research Program of multimedia convergence network on chip technology. EDA tools which were used in this work were supported by IC Design Education Center(IDEC).

References

[1] Laszlo Hars and Gyorgy Petruska, "Pseudorandom Recursions: Small and Fast pseudorandom Number Generators for Embedded Applications,"*EURASIP Journal on Embedded Systems*, Vol. 2007, No. 1 (2007), pp.5-5

[2] ISO/ISE 18000-6C : Parameter for air interface communications at 860Mhz to 960Mhz, 2005.

[3] Willick, D. L. et al, "Disk Cache Replacement Policies for Network Fileservers,"*Distributed Computing Systems*, 1993 Int'l Conf. pp. 2-11.

[4] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid configuration & instruction selection for an ASIP: A case study,"*Proc. Design Automation & Test Europe Conf.*, Mar. 2003, pp. 802–807.

[5] Narendra S. Bolabattin, "Random Number Generator Using Leap-Forward Techniques," 2005, http://www.pldesignline.com/showArticle.jhtml?articleID =192200271

[6] T. Glokler and H. Meyr, "Design of Energy-Efficient Application Specific Instructin-set Processors (ASIPs)," *Kluwer Academic Publishers*, pp. 117-143.

[7] G. Marsaglia, "DIEHARD: a battery of tests of randomness," 1996, http://stat.fsu.edu/pub/diehard