IEICE TRANSACTIONS

on Information and Systems

VOL. E108-D NO. 6
JUNE 2025

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Performance Enhancement of the LFSR-Based Unpredictable Random Number Generator in Rocket Core

Takayoshi SHIKANO[†], ***Student Member and*** **Shuichi ICHIKAWA**[†a)], ***Senior Member***

SUMMARY Masaoka et al. introduced an unpredictable random number generator (URNG) using a linear feedback shift register (LFSR) embedded within the CPU. Subsequent work by Kamogari and Ichikawa elucidated the LFSR requirements and the minimal essential period to pass the Diehard test. In this study we investigate a Rocket Core with a built-in LFSR, which was designed according to the results of preceding studies. By sampling the lower 32 bits of the 128-bit LFSR, a random number sequence was generated at a rate of 49.4 Mbit/s on a 50-MHz Rocket Core. The derived random sequence passed both the Diehard and NIST tests. Furthermore, we propose to replace an LFSR with a Leap-ahead LFSR, which applies its characteristic polynomial 32 times in a cycle. This improvement results in a significantly greater generation rate of 451 Mbit/s, while maintaining compliance with the Diehard and the NIST tests. The resource overhead of this URNG is negligible compared to the logic scale of the base system (LiteX/Rocket). Considering its low cost, high generation rate, high randomness quality, and ease of use, the proposed design is regarded to be a promising RNG support solution for a wide range of processors.

key words: *URNG, Embedded Processors, LFSR, RISC-V, FPGA*

1. Introduction

Random number generators are typically categorized into two types: True Random Number Generators (TRNG) and Pseudo Random Number Generators (PRNG). TRNGs generate random numbers from physical phenomena such as thermal noise or jitter, making their output difficult to predict. Meanwhile, a TRNG requires a special hardware to utilize physical phenomena, which results in a significant cost. PRNGs generate pseudo-random numbers using deterministic algorithms, which enables software implementations at a lower cost. A weakness of PRNGs is that their output can be predicted if their algorithm and internal state are uncovered.

Suciu et al. [1], [2] introduced an Unpredictable Random Number Generator (URNG), which utilizes the internal states of a CPU as its entropy source. A CPU is a complicated sequential logic circuit, whose internal states change each cycle. Suciu et al. utilized the performance counters as the entropy source, which are registers designed to monitor the performance and the internal states of a CPU through software. Since performance counters are widely integrated in commercial processors, the use of performance counters incurs no additional cost in many cases. However, performance counters typically contain a limited amount of entropy. It is thus difficult to generate high-quality random numbers using performance counters.

Masaoka et al. [3] presented a low-cost, high-quality URNG particularly suited for embedded systems, which integrates a Linear Feedback Shift Register (LFSR) into the CPU. LFSR is a kind of PRNG, whose future output can be predicted from its characteristic polynomial and its internal states. However, the external factors such as interrupts introduce fluctuations into the read-out intervals, which makes its output practically unpredictable. Masaoka et al. repeatedly read the value of the LFSR through software, and reported that the resulting number sequences successfully passed the Diehard test [4]. One of the problems of their work was a low generation rate (max 125 kbps). Another problem was that they did not use the NIST test [5], which is more stringent than the Diehard test.

Kamogari and Ichikawa [6] conducted a comprehensive simulation of Masaoka et al.'s method, and elucidated the design requirements of LFSR and the minimal essential period necessary to pass the Diehard test, assuming that the fluctuation of sampling intervals is uniformly distributed. Their findings indicate that the generation rate can be improved to approximately 100 times higher than that achieved by Masaoka et al., although this improvement is not yet confirmed through actual implementation. One of the purposes of the present study is to enhance the generation rate of LFSR-based URNGs and validate the upper limit of their performance on a hardware implementation.

In the present study, we conduct hardware implementations of LFSR-based URNGs following to the design guideline presented by Kamogari and Ichikawa. We then evaluate the randomness quality and generation rate of these implementations. For the quality evaluation, both the NIST test and the Diehard test are employed. Finally, the hardware resource utilizations are also shown to show the increase of cost is negligible.

This manuscript is organized as follows. Section 2 outlines the background of this work, and then Sect. 3 describes our implementation of Masaoka et al.'s URNG. Section 4 introduces a new design where an LFSR is replaced by a Leap-ahead LFSR to enhance the generation rate, together with the hardware resource utilization. Section 5 presents the advantages of our URNG in comparison to existing random number generators. Section 6 concludes the work.

This manuscript is based on the technical report by the authors [7], but is substantially enhanced and revised.

Manuscript received April 22, 2024.
Manuscript revised August 23, 2024.
Manuscript publicized December 10, 2024.

[†]Department of Electrical and Electronic Information Engineering, Toyohashi University of Technology, Toyohashi-shi, 441–8580 Japan.
a) E-mail: ichikawa@tut.jp

DOI: 10.1587/transinf.2024EDP7098

2. Background

2.1 LFSR

LFSR is a kind of PRNG, which is defined as a shift register whose input is determined by a linear function of its previous state. This linear function is also represented by a feedback polynomial (or characteristic polynomial) over Galois Field of order 2 (GF(2)). The cycle of LFSR reaches its maximal length if and only if the corresponding feedback polynomial is primitive over GF(2). The maximum period of an n-bit LFSR is $2^n - 1$, indicating that the internal state of LFSR traverses all possible values except zero. In Fig. 1, an example of 8-bit Fibonacci-type LFSR is illustrated. The feedback polynomial of this LFSR is represented as $x^8 + x^6 + x^5 + x^4 + 1$, where the input bit is generated by XORing the bits 8, 6, 5, and 4. In the following discussions, we represent this polynomial as a tap sequence [8, 6, 5, 4].

2.2 URNG by Masaoka et al.

Masaoka et al. [3] integrated an LFSR into the PULPino processor and generated random number sequence by reading the LFSR through software. Though an LFSR is a PRNG, the read-out interval is fluctuated by external interrupts and other factors. Thus, the derived sequence becomes practically unpredictable. The evaluation environment of Masaoka et al. is summarized in Table 1.

Though Masaoka et al. utilized the Diehard test to evaluate randomness quality, it is worth noting that the Diehard test itself does not define the specific evaluation criteria of various test results (p-values). Therefore, Masaoka et al. adopted the evaluation criteria of Fujieda et al. [8]. Firstly, each result is categorized into three classes (PASS / WEAK / FAIL) based on the criteria shown in Table 2. If many p-values are generated by a single test, Kolmogorov-Smirnov test (KS test) is applied on these p-values to verify that they are uniformly distributed. The result of this test is represented by the p-value of KS test. The Diehard test consists of 18 individual tests. If no FAIL is detected across all 18 tests, the random sequence is regarded to pass the Diehard test.

This present work also adopts the same evaluation criteria as Masaoka et al. Further details are found in Refs. [3], [8].

PULPino [9] is an open-source single-core microcontroller system, based on 32-bit RISC-V cores developed at ETH Zurich. RISC-V architecture defines Control and Status Registers (CSRs), accessible via csrr and csrw instructions. Masaoka et al. implemented a 128-bit LFSR and a 32-bit CSR in PULPino, where the CSR represents the least significant part of the LFSR. The random sequence was generated by reading the CSR through FreeRTOS [10].

The 128-bit LFSR is driven by system clock, and the value of CSR is iteratively read from software. The collected values were subsequently transmitted to the host computer and certified to pass the Diehard test. The measured generation rate was 125 kbit/s, which corresponds to the average sampling period of approximately 5000 cycles.

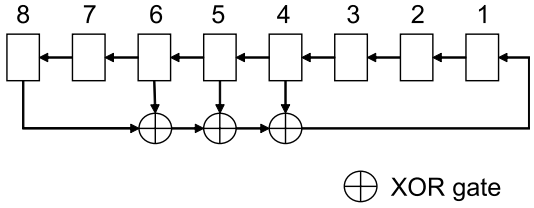

Fig. 1 An example of 8-bit LFSR [8,6,5,4]. [6]

Table 1 Experiment environment of Masaoka et al. [3]

Processor	PULPino [9] (RISCY Core)
Evaluation Board	AVNET ZedBoard AES-Z7EV-7Z020-G
Device	Xilinx XC7X020-CLG484-2
Target Freq.	20 MHz
OS	FreeRTOS v8.2.2 [10]

2.3 Analysis by Kamogari and Ichikawa

Kamogari and Ichikawa [6] conducted a comprehensive simulation using various LFSR configurations, including different lengths and tap sequences, to investigate the conditions for the Masaoka et al.'s URNG to pass the Diehard test. They observed that the presence of fluctuations in sampling intervals contributes to improve the randomness quality. Furthermore, they identified the following essential conditions to pass the Diehard test, when the tested sequence is generated under the worst-case condition (i.e., with NO fluctuations in sampling intervals).

- The taps of LFSR should be evenly distributed with four taps being sufficient.
- The sampling period should be 32 cycles or longer.
- The length of LFSR should be 48 bit or longer.

Kamogari and Ichikawa [6] summarized that the ideal generation rate can be estimated as f Mbit/s under a system clock frequency f MHz, considering 32 bits are generated every 32 cycles.

2.4 Leap-Ahead LFSR

Gu and Zhang [11] introduced a design to improve the generation rate of an LFSR by applying the feedback polynomial multiple times within a single cycle, and termed it Leap-ahead LFSR. Figure 2 (a) illustrates a conventional LFSR, which generates one bit in each clock cycle by applying the feedback polynomial once. In contrast, a Leap-ahead LFSR applies the feedback polynomial m times to generate m bits

Table 2 Diehard test evaluation criteria. [8]

Decision	Border by p-value
PASS	$0.005 \leq p < 0.995$
WEAK	$0.000001 \leq p < 0.005, 0.995 \leq p < 0.999999$
FAIL	$p < 0.000001, 0.999999 \leq p$

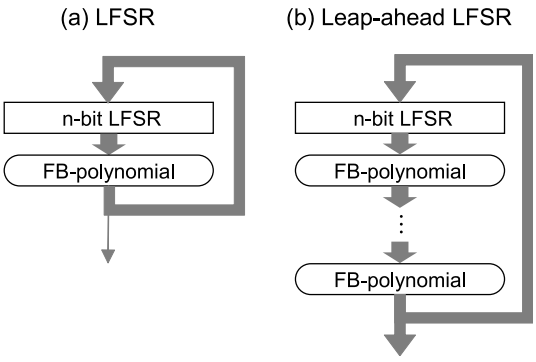

Fig. 2 LFSR and Leap-ahead LFSR. (©2023 IEEE) [12]

in each clock cycle (Fig. 2 (b)).

Ichikawa [12] investigated the randomness quality of various Leap-ahead LFSRs. According to Ichikawa's results, Leap-ahead LFSRs with a length of 44 bit or longer pass the Diehard test when $m \geq 32$, and the NIST test when $m = 64$.

3. LFSR-Based URNG

3.1 Deisgn Environment

Table 3 provides an overview of the implementation environment of this study. While Masaoka et al. adopted a small and lightweight platform for embedded systems, we adopted the hardware and software with higher performance and richer functionality to explore the high-performance and high-quality URNGs.

Our evaluation board, Arty A7-100T, contains a Xilinx Artix 7 FPGA, which is designed with Xilinx Vivado 2022.01 software. For the present study, we adopted LiteX [13] framework, which integrates the peripheral circuitry with the Rocket Chip. The Rocket Chip [14] is a parameterizable RISC-V processor developed at the University of California, Berkeley. It is an open-source processor configured with a 5-stage pipeline architecture. The source code is described in Chisel language [15] and can be translated into Verilog language. Table 4 summarizes the configuration of the Rocket Chip in this study.

Linux serves as the operating system on the FPGA chip, making the acquisition and processing of data much easier than the previous study by Masaoka et al. Users can interact with Linux from the console of the client PC, and transfer files between the client and the evaluation board via TFTP.

In the following evaluation, we integrate a 128-bit LFSR into the original Rocket Chip, with its least significant word (32 bit) implemented as a CSR for random number generation. The tap sequence of the 128-bit LFSR is set to [128, 77, 35, 11], as proposed by Živković [16]. The tap sequence [128, 7, 2, 1] of Masaoka et al. [3] was not adopted, since it is reported to yield low-quality random sequences [6]. FPU is not integrated in this implementation to reduce the logic scale. The system clock frequency is 50 MHz.

Table 3 Implementation Enviroment.

Processor	LiteX [13] with Rocket Chip [14]
Evaluation Board	Arty A7-100T
Device	Xilinx XC7A100TCSG324-1
Target Freq.	50 MHz
OS	Linux/riscv 6.0.0 Kernel Configuration

Table 4 Configuration of Rocket Chip.

Num. of core		1
ISA	64-bit RISC-V	
pipeline		5-stage
L1-cache	d-cache-block-size	64 byte
	d-cache-sets	64 byte
	d-cache-size	4096 byte
	d-cache-ways	2
	d-tlb-sets	1
	d-tlb-size	4
	d-tlb-ways	1
	i-cache-block-size	64 byte
	i-cache-sets	64 byte
	i-cache-size	4096 byte
	i-cache-ways	2
	i-tlb-sets	1
	i-tlb-size	4
	i-tlb-ways	1

```
#define read_csr(reg) ({ unsigned long __tmp; ¥
    asm volatile ("csrr %0, " #reg : "=r"(__tmp)); ¥
    __tmp; })
```

Fig. 3 Definition of read_csr function.

```
i = 0;
while(i < NUM){
    ret_lfsr_reader[i] = read_csr(0xca0);
    i++;
}
fwrite(ret_lfsr_reader, sizeof(unsigned int), NUM, fp);
```

Fig. 4 A part of C code that generates random numbers.

3.2 Random Number Generation

This subsection describes the evaluation process for random number generation and generation rate.

RISC-V architecture defines several extensions, one of which is Zicsr extension, i.e., Control and Status Register (CSR) Instructions [17]. The CSRR instruction, specified in Zicsr extension, serves to read the CSR described in the previous subsection. In C language, we define a macro function read_csr to embed a csrr instruction using an asm statement (Fig. 3). The volatile qualifier within the asm statement prevents the unintentional optimization during compilation process.

Figure 4 is a code snippet that reads the data from CSR and stores it to the corresponding array element. Each CSR instruction has a 12-bit address field that specifies a CSR, and the least significant word of the 128-bit LFSR is mapped to the address 0xca0. Once NUM words have been read, the content of the array is written to a file in binary format using the fwrite function. Although the execution time of fwrite

might be dependent on the buffer size, it was set to 1 MB in this instance.

Since the Diehard test requires approximately 100 MB of data, the parameter NUM was set to 3,000,000 here. The NIST test requires much larger data, while it is difficult to accommodate all data in the memory. To overcome this situation, we repeated the process shown in Fig. 4 to generate the data necessary for the NIST test.

The performance indices, the random number generation rate and the average sampling period, are calculated by the following equations.

First, the data size is calculated by Eq. (1). Each read_csr retrieves 32 bits (4 bytes) of data, which is repeated $NUM = 3 \times 10^6$ times.

$$DataSize[\text{byte}] = 4[\text{byte}] \times NUM \tag{1}$$

The random number generation rate, denoted as GenRate, is calculated by Eq. (2). To measure the CPU time, the Linux "time" command was utilized. The CPU time, denoted as CPUtime, is defined as the sum of "User Time" and "Sys Time" provided by the time command.

$$GenRate[\text{Mbit/s}] = \frac{8 \times DataSize[\text{byte}]}{10^6 \times CPUtime[\text{s}]} \tag{2}$$

The average sampling period, denoted as SmplPeriod, is given by Eq. (3). The average sampling time [s] is determined by dividing the data size of each sampling (32 bit) by the generation rate (bit/s), and then the average sampling period [cycle] is determined by multiplying the average sampling time [s] and the CPU clock frequency, denoted by ClkFreq [Hz]. The prefixes M in numerator and denominator cancel each other out.

$$SmplPeriod[\text{cycle}] = \frac{32[\text{bit}] \times ClkFreq[\text{MHz}]}{GenRate[\text{Mbit/s}]} \tag{3}$$

Kamogari and Ichikawa [6] noted that the randomness quality of the generated sequence might be degraded if SmplPeriod falls below 32 cycles. Although their estimation was made without the fluctuation of sampling periods, it is highly probable that the output sequence fails the randomness test when the SmplPeriod is smaller than 32 with minimal fluctuation.

3.3 Compiler Option

This subsection focuses on examining the impact of compiler options on both the randomness quality and the generation rate. Measurements were conducted with three optimize options: no option, -O option, and -O3 option. The iteration count NUM was set to 3,000,000 for the Diehard test.

The following two scenarios are considered.

1. In the first scenario (With fwrite), the generated random numbers are transmitted to the application located outside the FPGA. Here, the fwrite function is essential to send the data stored in the data array depicted in Fig. 4.
2. In the second scenario (Without fwrite), the generated random numbers are consumed within an application implemented in the FPGA. In this case, the execution time of fwrite must be excluded, as the loop control time and the CSR read time is intrinsic to this scenario.

Table 5 Generation rate (GenRate) and the average sampling period (SmplPeriod).

	Option	CPU Time [s]	GenRate [Mbit/s]	SmplPeriod [cycle]
With fwrite	(none)	4.36	21.01	76.2
	-O	2.52	36.43	43.9
	-O3	2.51	36.48	43.9
Without fwrite	(none)	2.87	31.90	50.2
	-O	0.20	451.1	3.55
	-O3	0.20	451.3	3.55

Table 6 Result of the Diehard test (Without fwrite).

Option	Trial	PASS	WEAK	FAIL
(none)	1	18	0	0
	2	16	2	0
	3	17	1	0
-O	1	6	2	10
	2	3	4	11
	3	5	2	11
-O3	1	5	3	10
	2	6	2	10
	3	5	2	11

Table 5 summarizes the CPU time, GenRate, and SmplPeriod for the abovementioned scenarios. The CPU time represents the average value of three trials, and GenRate and SmplPeriod are calculated based on this average CPU time.

In the second scenario, SmplPeriod decreases from 50 cycles (no option) to 3.55 cycles (-O and -O3). This corresponds to approximately 14-fold improvement in GenRate. However, the SmplPeriod of 3.55 cycles are much smaller than 32 cycles, which is the threshold to pass the Diehard test [6]. Thus, it becomes imperative to verify the randomness quality of the generated random sequences. It is noteworthy that even in the first scenario (With fwrite), the loop code to collect CSR values are exactly same as the second scenario.

Table 6 provides the summary of the Diehard test results on the random sequences generated in the second scenario (Without fwrite). To ensure reproducibility, the results of three trials are presented for each option. With no options, the generated sequences pass the Diehard test. Contrary, the generated sequences with -O and -O3 option fails. This is quite consistent to the results shown in Table 5. It is also evident that the quality of generated sequence is quite stable and reproducible.

Practically, the optimization in compilation phase is essential. It is strongly desired to achieve both the performance improvement with optimization options and the randomness quality of generated sequences.

While the measurement results shown in this subsection are consistent with the simulation results shown by Kamogari and Ichikawa [6], a more detailed investigation into the

```
i = 0;
while(i < NUM){
    ret_lfsr_reader[i] = read_csr(0xca0); // dummy
    ret_lfsr_reader[i] = read_csr(0xca0); // dummy
    ret_lfsr_reader[i] = read_csr(0xca0);
    i++;
}
```

Fig. 5 Generation rate control with dummy reads (#read_csr = 3).

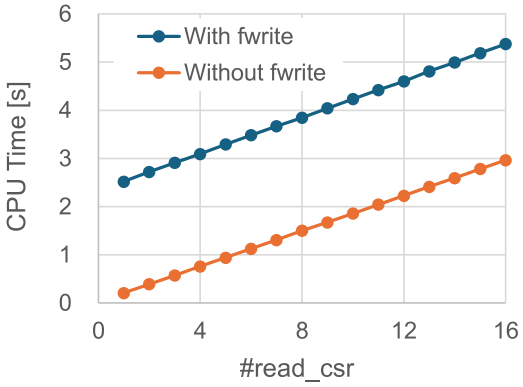

Fig. 6 #read_csr vs. CPU time [s].

tradeoffs between generation rate and randomness quality of generated sequences is necessary. Further experiments are conducted in the subsequent subsection.

3.4 Randomness Quality for Various Sampling Period

This subsection focuses on the relationship between randomness quality and average sampling period, where adjustments to the average sampling period can be made by inserting dummy read_csr calls into the loop.

Figure 5 presents an example code snippet, wherein the loop contains three read_csr calls. All three calls write into the same array element, which means that the first two calls are dummies to adjust the sampling intervals. In the subsequent discussion, "#read_csr" designates the number of read_csr calls within the loop. Despite modern compilers try to optimize code by eliminating the dummy codes, read_csr calls remain intact due to their volatile declaration, as shown in Fig. 3.

Figure 6 displays the measured CPU time for various #read_csr configurations. Despite compiling the code with -O option, the CPU time increases according to #read_csr, indicating that the dummy calls were not eliminated by the compiler.

Figure 7 summarizes the randomness qualities observed for various #read_csr values between 1 and 16. The Y-axis represents the number of FAILs in the Diehard test, denoted by #FAIL in the subsequent discussion. From the results of three trials plotted in Fig. 7, it is evident that the randomness quality exhibit consistent reproducibility.

In Fig. 7, it is notable that the randomness quality improves with increasing #read_csr. Specifically, the generated sequence passes all 18 tests when #read_csr > 9. However, in Trial 3, a single FAIL occurs when #read_csr is 14, where the

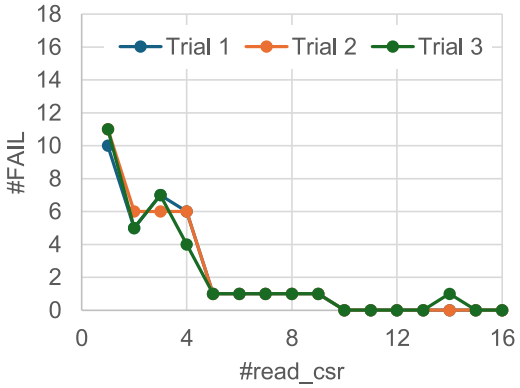

Fig. 7 #read_csr vs. #FAIL of the Diehard Test.

Table 7 GenRate and SmplPeriod (NUM=3,000,000, #read_csr=10).

	CPU Time [s]	GenRate [Mbit/s]	SmplPeriod [cycle]
With fwrite	4.23	21.65	73.9
Without fwrite	1.85	49.40	32.4

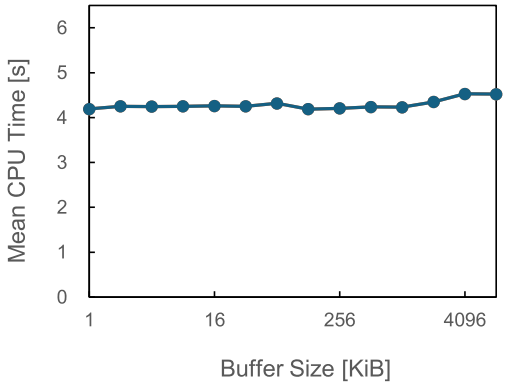

Fig. 8 Buffer size [KB] vs. CPU time [s] (with fwrite, #read_csr = 10).

failed test was OPERM5. Notably, OPERM5 of the Diehard test is known to be buggy [18], leading to many false positives in the preceding studies. Given that no FAILs were observed for #read_csr of 14 of Trials 1 and 2, it is reasonable to conclude that the FAIL of OPERM5 in Trial 3 for #read_csr of 14 is a false positive.

Table 7 summarizes the GenRate and SmplPeriod for #read_csr of 10. In the scenario "Without fwrite", SmplPeriod for #read_csr of 10 is 32.4 cycles, which is nearly comparable to the necessary condition to pass the Diehard test [6]. Under this condition, the GenRate is 49.4 Mbit/s, translating to 0.988 bits for each clock cycle. In other words, this achievement represents 98.8% of the maximum generation rate estimated by Kamogari and Ichikawa [6] was achieved. Furthermore, the generated random sequence successfully passed the NIST test [5] under this condition.

Turning to the other scenario "With fwrite", it is necessary to minimize the time required for fwrite function (cf. Fig. 6). Given that the size of buffer may influence the fwrite time, we conducted measurements of CPU times for various

buffer sizes[†]. However, the measurement results indicate that the buffer size has minimal impact on the CPU time in our present implementation (Fig. 8).

4. Leap-Ahead LFSR-Based URNG

In Sect. 3, we implemented Masaoka et al.'s URNG in a Rocket Chip, and confirmed that the random sequence generated at 98.8% of the estimated maximal rate successfully passed the Diehard and the NIST tests. However, in the implementation of the previous section, ad hoc adjustment of the code was essential to choose optimal number of dummy read_csr calls.

This section examines a new URNG configuration that generates high-quality random sequence at a high generation rate without any ad hoc code adjustments.

4.1 Generation Rate and Randomness Quality

One of the methods to remove dummy read_csr calls is to add hardware support to interlock the access to the LFSR. When two successive read_csr calls are performed, adequate number of wait cycles are inserted to the following call to keep the adequate intervals between two calls. By this control, the randomness quality of generated sequence is sustained by keeping average sampling period at an adequate level. No consideration is required on software side, and a simple code such as Fig. 4 can be always used.

Interlocking is a popular control in pipeline hardware to avoid hazards, and thus it is not difficult to implement interlocking in a processor. However, interlocking involves pipeline bubbles and naturally accompany performance degradation. Additionally, the control logic for interlocking may have negative impact to the operational frequency of the processor, which also leads to performance degradation. Another concern is the debug and verification efforts after modifying the control logic of the processor.

Therefore, we simply decided to replace an LFSR (Fig. 2 (a)) with a Leap-ahead LFSR (Fig. 2 (b)) to realize high generation rate and high randomness quality at the same time, without modifying the control logic. In the following discussion, NP represents the number of applications of characteristic polynomial in a single cycle.

Table 8 presents the results of the Diehard test conducted with the Leap-ahead LFSR. In this experiment, the LFSR in Rocket Chip was replaced by the Leap-ahead LFSR, and random sequences were generated using the code shown in Fig. 4 with the -O option. The length and tap sequence of the Leap-ahead LFSR remain the same as the original LFSR. Other parts of the implementation were left unchanged, resulting in identical execution cycles as shown in Table 5. Since the operational frequency was the same (50 MHz), the CPU times also remain the same as in Table 5.

In Table 8, NP=1 indicates the original LFSR, and thus the Diehard results are identical to those presented in Table 6 (-O). Though the design NP=1 fail the Diehard test, both the designs NP=7 and NP=32 successfully pass the Diehard test, achieving a generation rate of 451 Mbit/s and an average sampling period of 3.55 cycles.

Table 8 Diehard test results of various Leap-ahead LFSR designs.

Leap-ahead LFSR	Trial	PASS	WEAK	FAIL
128-bit (NP=1)	1	6	2	10
	2	3	4	11
	3	5	2	11
128-bit (NP=7)	1	18	0	0
	2	18	0	0
	3	17	1	0
128-bit (NP=32)	1	18	0	0
	2	16	2	0
	3	18	0	0

According to Kamogari and Ichikawa [6], the design NP=32 is anticipated to reliably pass the Diehard test, even when the CSR is read every cycles. In this sense, NP=32 is regarded as an assured and conservative design choice. However, the resource utilization of the Leap-ahead LFSR is expected to scale proportionally with NP. We thus explored the designs of smaller NP and found that the design NP=7 successfully passes the Diehard test. The corresponding SmplPeriod of NP=7 is $7 \times 3.55 \approx 25$ cycles, which fall shorter than the requirement (32 cycles) to pass the Diehard test. However, it should be reminded that the simulations by Kamogari and Ichikawa [6] was conducted with no fluctuation in sampling periods, and that the fluctuations enhance the randomness quality of generated sequences. In the design NP=7, it is regarded that various fluctuations appeared in an actual implementation, resulting in the success of the Diehard test at this short sampling period.

Furthermore, it is noteworthy that the generated sequences of NP=7 and NP=32 successfully passed the NIST test [5], which is more stringent than the Diehard test.

4.2 Resource Usage

Table 9 summarizes the resource usages of the designs listed in Table 8. The row labeled "Available" indicates the available number of units on the target device XC7A100T. The "baseline" design represents the original Litex/Rocket without the LFSR. As the results of CAD software may vary between trials, the results of three trials are shown for the designs NP=1, NP=7, and NP=32. Despite the expectation that a larger NP leads to a larger resource usage, the measured differences are not obvious and are almost negligible compared to the differences between trials. This observation suggests that the logic scale of the 128-bit LFSR or Leap-ahead LFSR is relatively small compared to the overall system framework (Litex/Rocket).

5. Discussion

The previous sections presented the implementation and evaluation of our LFSR-based URNG. This section highlights the advantages of our URNG in comparison to exist-

[†]The function setvbuf was used to change the buffer size.

Table 9 Hardware resources of the examined designs.

	Trial	Slice LUT	Slice Registers	Slice	LUT as Memory	LUT as Logic	BRAM	DSP	F7 Muxes	F8 Muxes
Available		63400	126800	15850	19000	63400	135	240	31700	15850
baseline	1	34592	17509	10394	3259	31333	29.5	15	1732	111
baseline	1	34706	17669	10429	3259	31447	29.5	15	1727	111
+128-bit LFSR (NP=1)	2	34701	17669	10084	3259	31442	29.5	15	1727	111
	3	34701	17669	10084	3259	31442	29.5	15	1727	111
baseline	1	34651	17669	10237	3259	31392	29.5	15	1732	111
+128-bit LFSR (NP=7)	2	34662	17669	10270	3259	31403	29.5	15	1732	111
	3	34662	17669	10270	3259	31403	29.5	15	1732	111
baseline	1	34720	17669	10414	3259	31461	29.5	15	1732	111
+128-bit LFSR (NP=32)	2	34712	17669	10462	3259	31453	29.5	15	1732	111
	3	34701	16669	10567	3259	31442	29.5	15	1732	111

Table 10 SHA and AES implementations for Artix-7 FPGA.

	Type	Speed grade	Max Freq. [MHz]	Throughput [Mbps]	LUT	FF	BRAM	Perf./Area [Mbps/Area]	Perf./Clock [Mbps/MHz]
Berten [21]	SHA-256	N/A	300	2300	3300	2500	0	5.05×10^{-1}	7.67×10^{0}
Xiphera [22]	SHA-256	N/A	177	28	1168	N/A	2	2.40×10^{-2}	1.58×10^{-1}
VISENGI [23]	AES 128	3	346	575	1113	585	1	4.09×10^{-1}	1.66×10^{0}
VISENGI [23]	AES 128	1	252	418	1112	586	1	2.98×10^{-1}	1.66×10^{0}
DornerWorks [24]	AES 128	3	333	42000	12000	7800	2	2.64×10^{0}	1.26×10^{2}
DornerWorks [24]	AES 128	1	270	8000	4700	4200	2	1.18×10^{0}	2.96×10^{1}
Xiphera [25]	AES 256	N/A	170	1360	4348	N/A	2	3.13×10^{-1}	8.00×10^{0}
This work	128-bit LFSR (NP=32)	1	50	451	143	128	0	2.18×10^{0}	9.02×10^{0}

ing random number generators. Since the characteristics of a URNG lie between those of a PRNG and a TRNG, we examine both in the following subsections.

5.1 Comparison with SHA and AES Core

Among various PRNGs, we focus on CSPRNG (Cryptographically Secure PRNG), since our URNG passes the randomness test such as the Diehard test and the NIST test. Various types of CSPRNG are specified in NIST SP 800-90A [20], which specifies mechanisms for the deterministic random bit generators (DRBG). Hash_DRBG and CTR_DRBG are widely acknowledged examples of DRBG, where well-established secure hash functions or block ciphers are utilized.

This section compares our URNG with SHA-256 (Secure Hash Algorithm) and AES (Advanced Encryption Standard) cores. Since the hash function or block cipher is the dominant component in Hash_DRBG or CTR_DRBG, evaluating SHA-256 or AES provides a reasonable approximation of these DRBGs. For a fair comparison, a 128-bit Leap-ahead LFSR (NP=32) was evaluated without interface logic and control logic.

Table 10 lists the specifications of SHA-256 and AES cores on Xilinx Artix-7 FPGA, which is the same platform used in our implementation. Although many commercial IP cores are available, few publish official results on the Artix-7, and some of the details are unavailable (e.g., FF in Xiphera [22], [25]). The columns Performance/Area and Performance/Clock were calculated by the authors, with the area estimated as $Area = LUT + FF/2$, considering that a single slice of Artix-7 contains 4 LUTs and 8 FFs[†].

Compared to two SHA-256 cores, our URNG is much smaller, more area-efficient, and more clock-efficient.

Compared to five AES cores, our URNG is smallest, second most area-efficient, and third most clock-efficient. While the faster version of DornerWorks [24] was the most area-efficient and clock-efficient, its area is 77 times larger than our URNG. Considering that the available number of LUTs are 20800 in XC7A35T and 63400 in XC7A100T, it is impractical for many applications. The smaller version of DornerWorks [24] is slightly more clock-efficient than ours, but our URNG is superior in other aspects.

In summary, our URNG is small and area-efficient. Additionally, it generates indeterministic output, which is generally harder to predict than pseudo-random numbers.

5.2 Comparison with TRNG

This section discusses the pros and cons of our URNG compared to various TRNGs. While there are many TRNG implementations, we specifically focus on those implemented on Xilinx Artix-7 FPGA, as in the previous section. For general information on the FPGA implementations of RNG, please refer to the survey by Bakiri [19].

Table 11 lists four TRNG implementations on the Artix-7 FPGA, each with different operation principles. Since TRNG circuits operate based on physical phenomena, they are not considered genuine digital circuits. Thus, the

[†]The Area for Xiphera [22], [25] was calculated as the number of FF is zero, which virtually serves as the lower bound of actual implementation.

Table 11 TRNG implementations for Artix-7 FPGA.

	Type	Speed grade	Throughput [Mbps]	LUT	FF	Perf./Area [Mbps/Area]
Fujieda and Ichikawa [8]	Latch metastability	1	20	716	974	1.66×10^{-2}
Fujieda [26]	TC-TERO	N/A	1.91	40	29	3.50×10^{-2}
Serrano et al. [27]	Multimodal RO	N/A	1.1 – 9.1	62	21	$1.52 \times 10^{-2} - 1.26 \times 10^{-1}$
Berten [28]	Multiphase oscillators	1	220	4600	5000	3.10×10^{-2}
This work	128-bit LFSR (NP=32)	1	451	143	128	2.18×10^{0}

Table 12 SHA and AES software implementations.

	command	Throughput [Mbps]
SHA-256	openssl speed -evp sha-256	4.77×10^{0}
AES 128	openssl speed -evp aes-128-ctr	5.36×10^{0}
This work		4.51×10^{2}

columns related to clocks have been omitted. The operational principles of four TRNGs are different. Fujieda and Ichikawa [8] utilized the metastability of latches, while the other three employed various types of ring-oscillators.

Compared to these four TRNG implementations, our URNG demonstrates the highest throughput and the highest performance per area. Since our URNG generates an unpredictable sequence that passes both the Diehard and the NIST tests, our URNG can serve as a high-performance and area-efficient alternative to a TRNG in many applications.

5.3 Comparison with Software

This section compares the performance of our URNG with software implementations of CSPRNG, specifically SHA and AES, as discussed in Sect. 5.1.

Among many implementations of SHA or AES, this study utilizes OpenSSL, which is widely adopted and well optimized for many platforms. The OpenSSL software library is a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication [29], and it includes a command interface to test library performance.

Table 12 summarizes the throughputs of SHA, AES, and our URNG on the evaluation system described in Table 3. Though openssl speed command reports the performances of various block sizes (16 - 16384 bytes), Table 12 presents the data of 16384 bytes. As readily seen, our URNG outperforms SHA and AES by factors of 95 and 84, respectively.

In summary, our URNG achieves 84 to 95 times higher throughput than software-based CSPRNG, with minimal hardware overhead.

6. Conclusion

This study presented a method for generating random number sequences, where software retrieves random numbers by reading a control register that implements the LFSR designed with appropriate parameters. When a 128-bit LFSR is implemented and its least significant 32-bit is provided as a CSR, the random sequence is generated at 49.4 Mbit/s with a 50-MHz processor. Importantly, the generated sequence passes both the Diehard and NIST tests. It should be noted that this design necessitates dummy instructions to optimize the sampling period for achieving its maximum generation rate.

Subsequently, this study proposed replacing the LFSR with the Leap-ahead LFSR to enhance the generation rate without compromising randomness quality. This new design further improves the maximum generation rate without dummy instructions. Our implementation achieves the generation rate of 451 Mbit/s, where the generated sequence successfully passes both the Diehard and NIST tests.

The LFSR and Leap-ahead LFSR utilized in this study are minimal in resource usage, making their implementation cost negligible compared to the resources of the entire system.

In summary, the proposed design can be integrated at a very low cost. Considering its high generation rate, high randomness quality, and ease of use, it is regarded to be a promising RNG support solution for a wide range of processors.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Number 20K11733 and 24K14878.

References

[1] A. Suciu, S. Banescu, and K. Marton, "Unpredictable random number generator based on hardware performance counters," Digital Information Processing and Communications, pp.123–137, Springer-Verlag, Berlin, 2011.

[2] K. Marton, A. Zaharia, S. Banescu, and A. Suciu, "Randomness Assessment of an Unpredictable Random Number Generator based on Hardware Performance Counters," ROMJIST, vol.20, no.2, pp.136–169, 2017.

[3] H. Masaoka, S. Ichikawa, and N. Fujioka, "Random Number Generation from Internal LFSR and Fluctuation of Sampling Interval," IEEJ Trans. Industry Applications, vol.141, no.2, pp.86–92, 2021. (in Japanese)

[4] G. Marsaglia, "Diehard battery of tests of randomness (Archived)," https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/, accessed Sept. 3, 2023.

[5] L.E. Bassham, A.L. Rukhin, J. Soto, J.R. Nechvatal, M.E. Smid, E.B. Barker, S.D. Leigh, M. Levenson, M. Vangel, D.L. Banks, N.A. Heckert, J.F. Dray, and S. Vo, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," NIST SP 800-22 (Rev.la), 2010.

[6] H. Kamogari and S. Ichikawa, "Evaluation of a Random Number Generator based on an Internal Linear Feedback Shift Register," IEEJ Trans. Industry Applications, vol.143, no.2, pp.87–93, 2023. (in Japanese)

[7] T. Shikano and S. Ichikawa, "Random number generation on the Rocket core with a built-in LFSR," IEICE Technical Report, vol.123, no.374, pp.1–6, 2024. (in Japanese)
[8] N. Fujieda and S. Ichikawa, "A latch-latch composition of metastability-based true random number generator for Xilinx FPGAs," IEICE Electron. Express, vol.15, no.10, Art no.20180386, May 2018.
[9] OpenHW Group, "pulp-platform/pulpino," GitHub Inc., https://github.com/pulp-platform/pulpino/tree/master/fpga, accessed Sept. 3, 2023.
[10] Amazon Web Services., "The FreeRTOS Kernel, Market Leading De-facto Standard and Cross Platform RTOS kernel," https://www.freertos.org/, accessed Sept. 3, 2023.
[11] X.C. Gu and M.X. Zhang, "Uniform Random Number Generator Using Leap Ahead LFSR Architecture," Proc. 2009 Intl. Conf. on Computer and Communications Security, pp.150–154, 2009.
[12] S. Ichikawa, "Pseudo-Random Number Generation by Staggered Sampling of LFSR," Proc. Eleventh Intl. Symp. on Computing and Networking (CANDAR 2023), pp.134–140, Nov. 2023.
[13] G.L. Somlo, "Toward a Trustable, Self-Hosting Computer System," 2020 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, pp.136–143, 2020.
[14] K. Asanović et al., "The Rocket Chip Generator," Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, USA, April 2016.
[15] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," Proc. 49th Annual Design Automation Conference (DAC 2012), San Francisco, CA, USA, ACM, pp.1216–1225, June 2012.
[16] M. Živković, "A table of primitive binary polynomials," Math. Comput., vol.62, no.205, pp.385–386, 1994.
[17] A. Waterman, K. Asanovic, J. Hauser, ed., The RISC-V Instruction Set Manual Volume II: Privileged Architecture, version 20211203, RISC-V International, Dec. 2021.
[18] R. Brown, "Robert G. Brown's General Tools Page," https://webhome.phy.duke.edu/˜rgb/General/dieharder.php, accessed Dec. 22, 2023.
[19] M. Bakiri, C. Guyeux, J.-F. Couchot, and A.K. Oudjida, "Survey on hardware implementation of random number generators on FPGA: Theory and experimental analyses," Computer Science Review, vol.27, pp.135–153, Feb. 2018.
[20] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," NIST SP 800-90A (Rev. 1), June 2015.
[21] Berten, "SHA-2 Hash Crypto Engine," BDS011 (v1.03), Feb. 2023.
[22] Xiphera, "XIP3327C: HKDF/HMAC/SHA-256/SHA-512 (Product Brief, ver. 1.0)," 20 Sept. 2023.
[23] VISENGI, "AES 128/192/256." Accessed: 9 Aug. 2024. [online]. https://www.visengi.com/products/aes
[24] DornerWorks, "AES-HS: High-Speed Encryption IP Core," 2017.
[25] Xiphera, "XIP1123B: Versatile AES-256 IP Core (Product Brief, ver. 1.0)," 20 Sept. 2023.
[26] N. Fujieda, "On the Feasibility of TERO-Based True Random Number Generator on Xilinx FPGAs," Proc. 30th Intl. Conf. Field-Programmable Logic and Applications (FPL 2020), Sweden, pp.103–108, 2020.
[27] R. Serrano, C. Duran, T.-T. Hoang, M. Sarmiento, K.-D. Nguyen, A. Tsukamoto, K. Suzaki, and C.-K. Pham, "A Fully Digital True Random Number Generator With Entropy Source Based in Frequency Collapse," IEEE Access, vol.9, pp.105748–105755, July 2021.
[28] Berten, "TRNG-P200 IP Core," BDS006 (v1.09), May 2024.
[29] OpenSSL, "Library." Accessed: 22 Aug. 2024. [online]. https://openssl-library.org/

Takayoshi Shikano received his B.E. degree in 2023 from the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. Presently, he is studying for his master's degree at that institution.

Shuichi Ichikawa received his D.S. degree in Information Science from the University of Tokyo in 1991. He has been affiliated with Mitsubishi Electric Corporation (1991-1994), Nagoya University (1994-1996), Toyohashi University of Technology (1997-2011), and Numazu College of Technology (2011-2012). Since 2012, he is a professor of the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. His research interests include parallel processing, custom computing machinery, and information security. He is a member of IEEE, ACM, IEICE, IEEJ, and IPSJ.