An HLS implementation of on-the-fly randomness test for TRNGs

Ryusei Oya and Naoki Fujieda
Department of Electrical and Electronics Engineering,
Faculty of Engineering,
Aichi Institute of Technology,
Toyota, Aichi, Japan
nfujieda@aitech.ac.jp (Naoki Fujieda)

Shuichi Ichikawa
Department of Electrical and Electronic
Information Engineering,
Toyohashi University of Technology
Toyohashi, Aichi, Japan
ichikawa@ieee.org (Shuichi Ichikawa)

***Abstract*—In this paper, we present an FPGA implementation of a randomness test called the count-the-ones test, which can be used to assure or control the quality of true random number generators (TRNGs) on-the-fly. Since our implementation adopts fixed-point arithmeric operations, reducing their precision is a trade-off between the amount of hardware and computational error. We used high level synthesis (HLS) to assess this trade-off easily. Based on our evaluation, we focused on two versions of circuits. A high-precision circuit required 607 LUTs and 536 FFs for 0.06% of an average error ratio of χ^2 value, whereas a resource-saving circuit reduced the amount of hardware to 377 LUTs and 430 FFs with 1.48% of an error ratio.**

I. Introduction

On the back of concern over security of digital systems, the importance of TRNGs (True Random Number Generators) has been grown. A TRNG provides a sequence of random numbers in an unpredictable way, according to a physical phenomenon. Particularly for a system based on an FPGA (Field-Programmable Gate Array) device, it is desirable to utilize logic [7], [14], clocking [4], and other elements of the FPGA to implement a TRNG.

A major concern for TRNGs is that the quality of their output random numbers may vary widely with operating conditions, individual difference of devices, and so on. It is known that attackers can lead a TRNG to failure by injecting frequencies into its power supply [9]. A sign of failure has to be detected to stop the random number generation or warn the system about it. Some types of TRNGs can control their behavior by a parameter input. They need an auto-calibration mechanism to find a proper parameter for each individual device and/or operating condition [1], [13]. Some other types of TRNGs are designed to improve its randomness at the cost of increasing power consumption or decreasing bit rate of generation. On-line quality control techniques have been recently proposed for them to make a better trade-off [2], [6]. To realize these quality assurance and/or control systems, an on-the-fly test circuit is essential to measure or estimate the quality of random numbers.

In this paper, we present an FPGA implementation of a statistical test called the count-the-ones test, which is included in the diehard battery of tests of randomness [10]. Although this test is much more complicated than other tests implemented by hardware, such as the monobit test and the runs test from the NIST test suite [15], it can detect flaws of random numbers that cannot be detected with simple tests. Our implementation is optimized for block memory and DSP (Digital Signal Processing) units of FPGA to minimize the required amount of hardware. Since our implementation adopts fixed-point arithmeric operations, reducing their precision is a trade-off between the amount of hardware and computational error. To easily assess it, we coded the test in C++ and synthesized circuits using the Xilinx's Vitis HLS (High-Level Synthesis) tool [19]. We evaluate the error ratio and the amount of hardware with various set of precisions.

The rest of this paper is organized as follows. In section II, we briefly introduce hardware implementations of randomness tests and existing quality assurance and control systems for TRNGs. The algorithm of the count-the-ones test is explained and analyzed in Section III, and the proposed hardware implementation is described in Section IV. Section V presents an evaluation about the error ratio and the amount of hardware of the synthesized circuits. We conclude the paper in Section VI.

II. Background

A. Hardware Implementations of Randomness Tests

The output sequence of random numbers from a TRNG must be unpredictable and, at least, free from regularity. To confirm that there is no evidence of regularity found from them, various randomness tests have been proposed and used. Since regularity may appear from different aspects, test suites have also been proposed by combining a number of tests with different characteristics. Famous test suites include the diehard test [10] and the NIST SP 800–22 test suite [15]. Sets of tests defined in standards such as AIS–31 [8], FIPS 140–2 [12] and NIST SP 800–90B [17] have also been used.

The basis of randomness tests is hypothesis testing. There, the null hypothesis is set that ones in the given bit sequence appear with the probability of $1/2$ and they are independent and identically distributed. A statistical value is calculated from the bit sequence with a certain algorithm. The probability of obtaining a value at least as extreme as the observed value, assuming that the null hypothesis is correct, is defined as the p-value. If the p-value becomes extremely low, the null hypothesis is rejected, meaning that the bit sequence has regularity.

In 2009, Santoro *et al.* first presented a hardware implementation of randomness test for checking the output bit sequence of a TRNG on-the-fly [16]. They implemented a variant of the Maurer's universal test [11], which is also included in NIST test suite (as the universal test) and AIS–31 (as the Test T8). They discussed an approximation of the logarithm function, required to estimate the entropy of the bit sequence.

Since then, various hardware implementations of tests have been proposed. Most of the implemented tests were relatively simple tests, included in NIST SP 800–22 and FIPS 140–2, in consideration of the required amount of hardware. Although some types of regularity can be detected only by tests in the diehard test, they have been seldom implemented by hardware because of their complexity.

To the best of our knowledge, Vaskova *et al.* have presented the only hardware implementation of tests from the diehard test in the past [18]. They selected four tests: the bitstream test, the OPSO test, the OQSO test, and the DNA test, which are based on the same algorithm, and proposed an efficient usage of hardware. However, the goal of their research is to accelerate time-consuming randomness tests. The amount of hardware of their implementation was too large to be used for on-the-fly testing of a TRNG.

In this paper, we select another test from the diehard test, called the count-the-ones test. As we will describe later in Section IV and Section V, this test can be effiently implemented by hardware and it requires only hundreds of LUTs (look-up tables) and FFs (flip-flops), by allowing computational error due to fixed-point arithmetic operations.

B. Quality control systems for TRNGs

More recently, on-line quality control mechanisms for TRNGs have been proposed. Most of the implementations of randomness tests presented in Section II.A are used for quality assurance: they detect a flaw from the output of TRNG to stop the generation or alarm the system. In addition, quality control systems can find an appropriate parameter of the target TRNG and/or pursue a better trade-off between the quality of generated random numbers and the cost of generation (e.g., increase of power consumption).

Carriera *et al.* presented a control system for an on-line parameter search for TRNG [1]. The target TRNG is based on a transition effect ring oscillator and its behavior can be changed by a parameter [21]. It gives biased output in case of taking too long time to generate random bits. To find an appropriate parameter, the system has to check the number of clock cycles for generation using a counter, in addition to its output using basic randomness tests. The idea of estimating entropy not only from output random bits but also from the behavior of ring oscillators is also seen in other control systems [2], [13].

Gonzalez *et al.* [6] proposed a quality control method for two types of TRNGs, multiple ring oscillators and a self-timed ring, to balance the randomness quality and power consumption. By decreasing the number of ring oscillators or the number of stages of the self-timed ring, the power consumption can be reduced in exchange for the loss of entropy. With a set of simple randomness tests [20], the system checks if the output bit sequence has entropy higher than a certain threshold. The threshold is determined depending on the intended use of the random numbers. This enables to reduce power consumption of the TRNG in some applications. Another system presented by Gantel *et al.* [5] used the results of on-line randomness tests to determine if the output of the TRNG needs to be post-processed.

We think that the proposed implementation of the count-the-ones test becomes a useful component of quality control systems. In these systems, tests were often selected according to the characteristics of the target TRNG and used complementarily. They might expect test circuits to be implementable with minimum amount of hardware in some cases, rather than to have high precision of test results. Our implementation is based on HLS, in order to make it easy to assess the trade-off between amount of hardware and precision.

III. Analysis of Count-the-one's Test

A. Description of the algorithm

Fig. 1 abstracts the procedure of the count-the-ones test. The test requires 2,048,032 bits (256,004 bytes) of random numbers. It consists of three steps: 1) initialization of arrays (lines 1–6), 2) count of the occurrence frequency of each word (lines 7–15), and 3) calculation of the difference of χ^2 values (lines 16–26). The p-value is calculated from the difference, denoted by $chsq$ in Fig. 1.

In the second step, each input byte is first translated into an alphabet, according to the number of ones in the byte (line 8). Table I depicts the mapping of alphabets, along with the occurrence probability of each alphabet, assuming that ones appear with the probability of $1/2$, and internal encoding in

```
for $str4$ in 4-letter words do
    $freq4[str4] \leftarrow 0$
end for
for $str5$ in 5-letter words do
    $freq5[str5] \leftarrow 0$
end for
for $i$ in 0 to 256003 do
    $letter_i \leftarrow$ byte_to_letter($rand\_bit_{8i:8i+7}$)
    if $i \geq 5$ then
        $str4 \leftarrow letter_{i-3:i}$
        $str5 \leftarrow letter_{i-4:i}$
        $freq4[str4] \leftarrow freq4[str4] + 1$
        $freq5[str5] \leftarrow freq5[str5] + 1$
    end if
end for
$chsq \leftarrow 0$
for $str4$ in 4-letter words do
    $ev \leftarrow$ expected_freq($str4$)
    $ov \leftarrow freq4[str4]$
    $chsq \leftarrow chsq - (ov - ev)^2/ev$
end for
for $str5$ in 5-letter words do
    $ev \leftarrow$ expected_freq($str5$)
    $ov \leftarrow freq5[str5]$
    $chsq \leftarrow chsq + (ov - ev)^2/ev$
end for
$z \leftarrow (chsq - 2500)/\sqrt{5000}$
$p \leftarrow$ erfc($z/\sqrt{2}$)/2
```

Fig. 1. Pseudocode of the count-the-ones test.

TABLE I
MAPPING OF ALPHABETS IN THE COUNT-THE-ONES TEST.

# of ones	0–2	3	4	5	6–8
Alphabet	A	B	C	D	E
Probability	37/256	56/256	70/256	56/256	37/256
Encoding	`000`	`010`	`100`	`011`	`001`

our implementation. It then forms 256,000 [1] overlapping 4-letter ($str4$) and 5-letter ($str5$) words and counts the occurrence frequency of each word (lines 10–13).

The third step calculates the difference between the χ^2 value for 4-letter words (lines 17–21) and that for 5-letter words (lines 22–26). The expected frequency (expected_freq) of an n-letter word $w_{0:n-1}$ is calculated by

$$256000 \times \prod_{i=0}^{n-1} \mathrm{prob}(w_i), \tag{1}$$

where the function prob returns the occurrence probability of the letter, shown in Table I. The resultant value, $chsq$, follows a normal distribution with a mean of 2,500 and standard deviation of $\sigma = \sqrt{5{,}000}$.

B. Considerations on Hardware Implementation

To reduce the amount of hardware, we consider the use of RAMs, ROMs and fixed-point arithmetic operations in our implementation. Occurrence frequencies of words ($freq4$ and $freq5$) can be stored in respective RAMs. The byte_to_letter and expected_freq functions can be replaced with ROMs because their outcomes can be pre-calculated. To avoid a division with the expected frequency (lines 20 and 25 in Fig. 1), reciprocals of expected_freq are also pre-calculated and stored in a ROM.

Fixed-point arithmetic is applied to the calculation of the $chsq$ value. More specifically, the variables ev and $chsq$ are stored as fixed-point numbers. Note that the variable ov, the occurrence frequency of a word, is an integer. Now that we do not have to consider loss of significance on floating-point arithmetics, we reverse the order of addition (lines 22–26) and subtraction (lines 17–21) in order to let $chsq$ be an unsigned value. Since the bit widths of the fractional portions of the variables affect the computational precision, we let them be parameters of our implementation and assess a trade-off with the amount of hardware in Section V.

The bit widths of the integer portions should be determined in order to avoid overflow of the $chsq$ value except in an extreme case. If an overflow is detected, we terminate the calculation and output the p-value of zero. It is common that a TRNG with an improper setup outputs zeros or ones constantly. This means all of the 5-letter words become “AAAAA” or “EEEEE” and the $chsq$ value reaches about 2^{32}. However, it is nonsense to set a large bit width for this case because the p-value will be approximated as zero after all. We set the width of the integer portion of $chsq$ as 16 bits and the threshold for overflow detection as 2^{15} heuristically. An overflow does not occur with this threshold if the final $chsq$ value is less than 10,000.

In addition, the calculation of z and p-value (lines 27–28 in Fig. 1) can be excluded from hardware implementation. As we have explained in Section II.B, a randomness test in a quality control system is used to check if the estimated entropy or p-value is within a certain range (like a system proposed by Gonzales *et al.* [6]). Such a check can be done with the $chsq$ value. For example, the condition that the p-value is 0.001 or higher is approximately equivalent to that the $chsq$ value is smaller than 2,718.5 (3.09σ higher than the mean).

IV. Hardware Implementation

A. Dataflow of Calculation

Figure 2 depicts a block diagram of our implementation of the count-the-ones test. Note that initialization of the $freq4$ and $freq5$ arrays is omitted from the figure. Pipeline registers inserted automatically by an HLS tool are not considered. RAMs and ROMs are shown in gray. The diagram has three sections: calculation of the indices, increment of the counters, and calculation of the $chsq$ value. The second step in Section III.A includes the first and second sections, while the third step correspond to the third section.

First, the indices of RAMs ($index4$ and $index5$) are calculated in the first section and the corresponding counters are incremented in the second section, in a pipelined structure. Since $index4$ and $index5$ become 4-digit and 5-digit quinary numbers, the numbers of the counters in the RAMs is

[1] Due to a bug of the original implementation of the diehard test, the actual number of words is 255,999 and the first byte is never used.

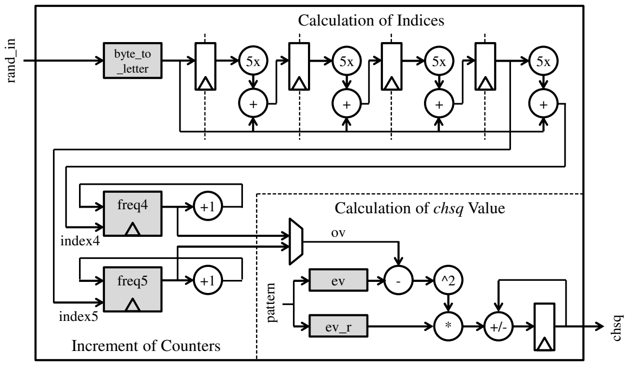

Fig. 2. Block diagram of proposed implementation of the count-the-ones test.

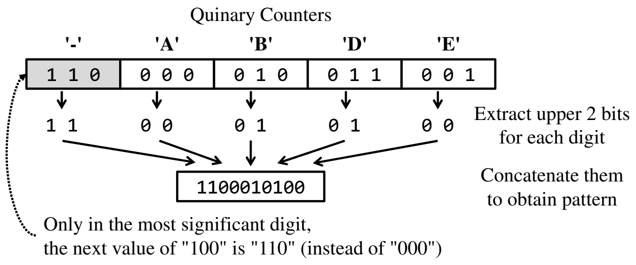
Quinary Counters
'-'
'A'
'B'
'D'
'E'
1 1 0
0 0 0
0 1 0
0 1 1
0 0 1
1 1
0 0
0 1
0 1
0 0
Extract upper 2 bits for each digit
Concatenate them to obtain pattern
1100010100
Only in the most significant digit, the next value of "100" is "110" (instead of "000")

Fig. 3. Generation of the ev and ev_r ROM address.

$5^4 = 625$ and $5^5 = 3{,}125$, respectively. This pipeline proceeds once in 3 cycles because a counter in the RAMs must be read, incremented, and written.

The $chsq$ value is calculated in the third section. A counter value is read from one of the RAMs as ov. The expected value of the corresponding word and its reciprocal are read from ROMs as ev and ev_r, respectively. The result of calculating $(ov-ev)^2 \times ev\_r$ is added to or subtracted from the $chsq$ value, according to which RAM was read. After the accumulation for all of the words, the circuit halts and outputs the final $chsq$ value.

The number of cycles to complete the test in this implementation is estimated at 774,887. The initialization step will take 3,125 cycles to reset counters in the $freq5$ RAM. The counting step will take 768,012 cycles as it requires 3 cycles for each of 256,004 bytes. The calculation step will take 3,750 cycles to read counter values in the $freq4$ and $freq5$ RAMs. This means that the proposed implementation does not become a performance bottleneck. The throughput of a TRNG is usually much smaller than 2.64 bits/cycle ($\simeq$ 256,004 $\times$ 8/774,887).

B. Size of RAMs and ROMs

In this section, we describe an estimation of the number of RAM blocks and an optimization of ROMs to reduce it. The target FPGA SoC of our implementation, Zynq-7000, has 36-kbit RAM blocks. A 18-kbit RAM can also be implemented with a half of a RAM block. We count it as 0.5 RAM blocks.

The byte_to_letter ROM is not expected to be implemented with RAM blocks because the size of this ROM is only 768 ($= 3 \times 2^8$) bits. It will be implemented with LUTs.

```
ap_ufixed<CHSQ_T,CHSQ_I> chsq;
ap_ufixed<EV_T,EV_I> ov, ev;
ap_ufixed<CHSQ_F,0> ev_r;
ap_ufixed<EV_T*2,EV_I*2> diffsq;
ap_ufixed<(EV_T*2)+CHSQ_F,EV_I*2> chsq_inc;
```

Fig. 4. Definition of fixed-point variables in the C++ code.

```
ev = evalue[pattern];
ev_r = evalue_rev[pattern];
if (i < 3125) {
    ov = freq5[index5++];
} else {
    ov = freq4[index4++];
}
ev = (ev > ov) ? ev - ov : ov - ev;
diffsq = ev * ev;
chsq_inc = diffsq * ev_r;
```

Fig. 5. Calculation of `chsq_inc` in the C++ code.

The RAMs to count the occurrence frequencies of words use 2.5 RAM blocks. Since the number of 4-letter and 5-letter words is $5^4 = 625$ and $5^5 = 3{,}125$, the number of words of the $freq4$ and $freq5$ RAMs becomes 2^{10} and 2^{12}, respectively. A value of each counter is stored as a 13-bit integer and it saturates at 2^{12}. This saturation does not occur as long as an overflow of the $chsq$ value (explained in Section III.B) is avoided. Thus, the $freq4$ and $freq5$ RAMs require 0.5 and 2 RAM blocks, respectively.

The address input of the ev and ev_r ROMs can be 10 bits wide with a careful encoding of the alphabets. Figure 3 describes how to generate the address input. Along with the indices of the RAMs ($index4$ and $index5$), the current word in the $chsq$ calculation is managed by a 5-digit quinary counter. The address input is obtained by concatenating upper two bits of respective digits. As described in Table I, ‘A’ and ‘E’, ‘B’ and ‘D’ have the same occurrence probability, respectively. We encode the alphabets in order that these pairs have the upper two bits in common. This enables to equate alphabets in each pair in the calculation of occurrence probability of words. When dealing with 4-letter words, the most significant digit is set to “110” because the upper two bits of “11” are not used by any alphabet. As a result, each of these ROMs is 2^{10} words deep and fit in 0.5 RAM blocks if each of the words (i.e., ev and ev_r) is no more than 18 bits wide.

In summary, we can estimate that our implementation uses 3.5 RAM blocks in total (2.5 for the RAMs of $freq4$ and $freq5$ and 1.0 for the ROMs of ev and ev_r), if the ROMs are kept small enough.

C. Coding in C++

We use the fixed-point arithmetic library of the Vitis HLS tool (`ap_fixed.h`), which defines a type for unsigned, fixed-point numbers as `ap_ufixed`. It is actually a C++ template that takes two non-negative integers. A `W`-bit fixed-point variable with `I` bits of integer portion can be defined with the `ap_ufixed<W, I>` type.

Figure 4 enumerates the definition of fixed-point variables in our code, while Fig. 5 describes the process of calculating the value to be incremented to or subtracted from $chsq$. The variables `diffsq` and `chsq_inc` correspond to $(ev-ov)^2$ and $(ev-ov)^2 \times ev\_r$, respectively. The bit widths of $chsq$ and ev are defined as constants in a header file. Constants with the prefix of `_T` are the total bit width of variables, while those with `_I` and `_F` are the widths of the integer and the fractional portions, respectively. Since ev_r, or a reciprocal of expected occurrence frequency, is always smaller than 1, its integer portion is not needed. The bit width of a product is calculated by the sum of widths of operands.

After that, arithmeric operators can be used as usual, as described in Fig. 5. The variable `pattern` is the address of the ROMs, generated by the process shown in Fig. 3. The arrays `evalue` and `evalue_rev` correspond to the ev and ev_r ROMs, respectively.

V. Evaluation

A. Methodology

To evaluate a trade-off between the amount of hardware and computational error, we synthesized the proposed implementation of the count-the-ones test, using various bit widths of the fractional portion of fixed-point variables. The synthesis tools are Vitis HLS 2021.1 (for high level synthesis) and Vivado 2021.1 (for logic synthesis). The target FPGA SoC device is XC7Z020-CLG400-1. The target clock frequency is set to 100 MHz. On exporting the design as an Vivado IP core using the `export_design` command, we have the core be logic synthesized by adding a `-flow syn` option. The input random numbers are passed with an AXI-Stream interface, while the output $chsq$ value is made available as a memory-mapped register of an AXI-Lite interface.

The width of the fractional portion of the $chsq$ value was set from 8 to 24 with an interval of 2. The fractional portion of ev was from 0 to 8 bits wide, also with an interval of 2. The numbers of LUTs, FFs, RAM blocks (BRAMs), and digital signal processing units (DSPs) required for the core were recorded.

The dataset of random numbers was obtained from an existing TRNG [3]. We coded a floating-point version of the test for reference. Fifty sets of 2-Mbit random numbers that give $chsq$ values of less than 10,000 in the floating-point version were selected. The error ratio was calculated with the following formula:

$$\frac{|chsq_I - chsq_F|}{chsq_F} \times 100[\%], \tag{2}$$

where $chsq_I$ and $chsq_F$ are the $chsq$ values in the fixed-point version and the floating-point version, respectively.

B. Operation of Circuit

According to an estimation from a synthesis report of Vitis HLS, the synthesized IP core takes 774,907 cycles to complete the test. The cycle count fluctuated one or two cycles probably due to optimizations. It was very close to our estimation

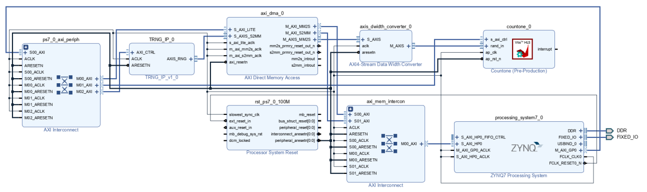

Fig. 6. Block diagram of a system with the synthesized IP core.

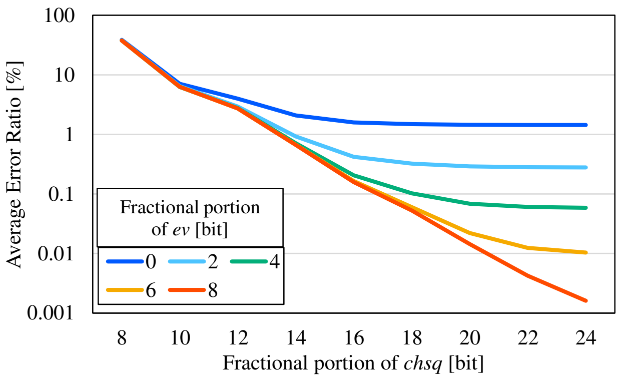

Fig. 7. Average error ratio with fixed-point arithmetics.

shown in Section IV.A. This means that our C++ code was synthesized as we have expected. This also means that the throughput of the circuit will be 264 Mbit/s (See Section IV.A), much higher than most of the TRNGs for FPGAs.

To verify the operation of the synthesized IP core, we constructed a system that includes one of the synthesized cores. Figure 6 depicts the block diagram of the system. The upper-right block with a red icon is the synthesized core. The system uses the direct memory access (DMA) core in two ways: providing random numbers stored in the main memory with the synthesized IP core, and storing random numbers from a TRNG to the main memory. We confirmed that the same output value as C simulation was obtained in a real machine.

C. Error Ratio

Figure 7 plots the average error ratio from the floating-point version. The X-axis represents the bit width of the fractional portion of the $chsq$ value, while the Y-axis is the average error ratio with fifty sequences of random numbers. Each line corresponds to the same width of the fractional portion of the ev value.

The graph implies that the error comes from truncation of both values and their effects are almost independent. When the width of the fractional portion of $chsq$ is increased, the average error ratio saturates at a certain value, according to the width of the fractional portion of ev. It is also observed that 0.1% of the error ratio was achieved when the widths were set to 18 bits for $chsq$ and 4 bits for ev. In Section V.D, we compare the amount of hardware around this point.

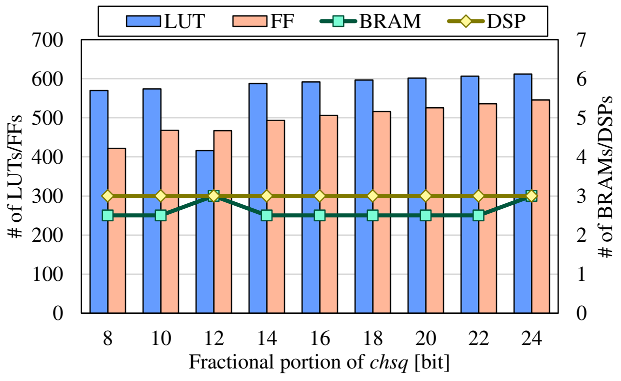

Fig. 8. The number of FPGA elements with various widths of $chsq$.

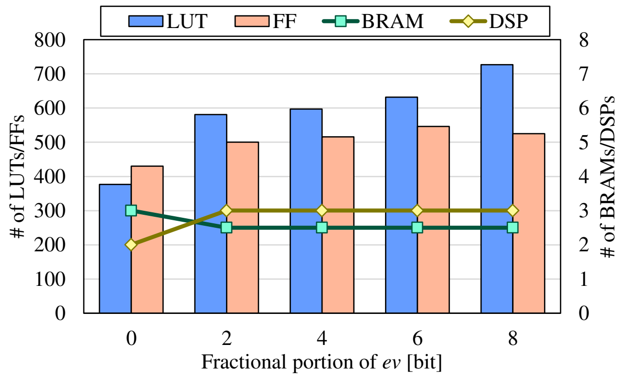

Fig. 9. The number of FPGA elements with various widths of ev.

D. Amount of Hardware

Figure 8 summarizes the number of FPGA elements (LUTs, FFs, BRAMs, and DSPs) required for the synthesized IP core. The X-axis stands for the bit width of the fractional portion of $chsq$. The fractional portion of ev was fixed to 4 bits wide. The numbers of LUTs and FFs are shown in bars and correspond to the left axis, while BRAM and DSP counts are shown in lines and correspond to the right axis. Note that these results include the AXI-Stream and AXI-Lite interfaces, which approximately corresponds to 50 LUTs and 100 FFs, in addition to the computation circuit itself (shown in Fig. 2)

The number of RAM blocks was 2.5 in most cases, which was smaller than our estimation of 3.5 (see Section IV.B). Vitis HLS also estimated it at 3.5. According to logic synthesis reports from Vivado, there are two reasons. First, the RAMs of $freq4$ and $freq5$ were optimized to use 2 RAM blocks. Second, the ROM of ev was not implemented with a RAM block, but with LUTs. Since we did not give a directive to use RAM blocks, the tool presumably made the decision to use LUTs, taking into account the balance of the elements used. Nevertheless, it was sometimes implemented with a RAM block, in the case of 12 bits for instance. In this case, the number of LUTs was decreased by about 160 but the number of RAM blocks was increased by 0.5. An increase of RAM blocks in the case of 24 bits was simply because the ROM did not fit in a single 18-kbit RAM block.

Figure 9 describes the relationship between the precision of ev and the amount of hardware. The X-axis represents the width of the fractional portion of ev in this figure. The fractional portion of $chsq$ was 18 bits wide.

When ev is treated as an integer (i.e., its fractional portion is 0 bits wide), the number of DSP units was reduced to 2. In this case, $(ov-ev)^2$ becomes 24 bits wide and the subsequent multiplication by ev_r fits in a single DSP. A DSP unit in Zynq-7000 FPGA SoC have a 25-bit $\times$ 18-bit multiplier. This changed the balance of the elements used: the ROM of ev was always implemented with RAM blocks in this case. It is also observed that the number of LUTs started to increase rapidly when the width was increased to 6 bits or more. We think that it comes from the increase of the bit width of the multiplications.

TABLE II
BREAKDOWN OF AMOUNT OF HARDWARE.

Part	resource-saving		high-precision	
	LUT	FF	LUT	FF
Total	377	430	607	536
Step 1	50 (13%)	14 (3%)	52 (9%)	14 (3%)
Step 2	152 (40%)	141 (33%)	150 (25%)	141 (26%)
Step 3	117 (31%)	160 (37%)	347 (57%)	266 (50%)
AXI-Lite	31 (8%)	85 (20%)	31 (5%)	85 (16%)
AXI-Stream	21 (6%)	20 (5%)	21 (3%)	20 (4%)
others	6 (2%)	10 (2%)	6 (1%)	10 (2%)

E. Candidate Circuits

According to the evaluation results, we selected two circuits as candidates for being used in quality control systems for TRNGs: a high-precision circuit and a resource-saving circuit. In the high-precision circuit, the widths are set to 22 bits for $chsq$ and 4 bits for ev. It shows an excellent error ratio of 0.06%. The numbers of LUTs and FFs required are 607 and 536, respectively. The widths of the resource-saving circuit are 18 bits for $chsq$ and 0 bits for ev. If an increased error ratio of 1.48% is acceptable, the amount of hardware can be reduced to 377 LUTs and 430 FFs with this circuit. On the selection of the circuits, we first selected the width of fractional portion of ev according to Fig. 9, which has a greater impact on the amount of hardware. The width of $chsq$ for each circuit was then selected from Fig. 7, at the point where the average error ratio saturates.

Table II compares the breakdown of amount of hardware between the selected circuits. A number in parentheses indicates the percentage of each module to the total. Steps 1–3 stand for the steps of the algorithm described in Section III.A, which were synthesized as separated modules. As expected, the diffrence only appeared in Step 3, the calculation of the $chsq$ value. The reduction of the number of LUTs there was mostly due to the replacement of a ROM with a RAM block as we have described above. The reduction of FFs came from the reduction of pipeline registers, due to not only the decrease of the bit width of calculation, but also the decrease of the number of pipeline stages.

TABLE III
COMPARISON OF FPGA IMPLEMENTATIONS OF RANDOMNESS TESTS.

Authors	Test	Family	LUT	FF	DSP	length [bit]
Santoro *et al.* [16]	Universal [8]	Virtex-5	1,634	n/a	n/a	2,068,480
Vaskova *et al.* [18]	Bitstream/OPSO/OQSO/DNA [10]	Virtex-5	15,375	8,222	n/a	2,097,171
Yang *et al.* [20]	Entropy Estimation [17]	Spatran-6	174	81	1	8,192
Gantel *et al.* [5]	Chi-square	Kintex-7	274	260	1	512
Gantel *et al.* [5]	Repetition Count [17]	Kintex-7	69	110	0	$\sim$50
Gantel *et al.* [5]	Adaptive Proportion [17]	Kintex-7	192	30	0	1,024
This work	Count-the-ones [10]	Zynq-7000	373	430	2	2,048,032

Finally, we made a comparison of the resource-saving candidate circuit with other FPGA implementations of randomness tests. Table III summarizes the comparison result. The rightmost column stands for the length of the input random numbers in bit. The most important difference of the proposed implementation is that the amount of hardware (LUTs and FFs) is much smaller than the past implementations of complicated tests [16], [18]. They could detect a long-term defect of random numbers while they required a large number of logic elements. In contrast, simple tests can be implementable with hundreds of LUTs and FFs but detect only a short-term loss of randomness. We successfully implemented a complicated test with a hardware cost not much different from simple tests. The cost is not much larger than recent TRNGs for FPGA [14], either. This makes it easy to integrate the proposed circuit to quality control systems, which take both short-term and long-term tendencies of random numbers into account.

VI. CONCLUSION

We presented an HLS implementation of the count-the-ones test from the diehard test, for on-line quality control of a TRNG on an FPGA. We also assessed a trade-off between the amount of hardware and computational error and proposed two circuits as candidates for being used in quality control systems.

We are going to develop an on-line quality control system using the proposed implementation. If needed, we will implement other types of randomness tests in a similar way to what we did in this paper.

ACKNOWLEDGMENT

A part of this study was supported by JSPS Grants-in-Aid for Scientific Research (KAKENHI) Grant Number 20K11733 and 21K12164.

REFERENCES

[1] L. B. Carreira *et al.*, "Low-Latency Reconfigurable Entropy Digital True Random Number Generator With Bias Detection and Correction," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 5, pp. 1562–1575, 2020.

[2] T. Chen *et al.*, "A Lightweight Full Entropy TRNG With On-Chip Entropy Assurance," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 12, pp. 2431–2444, 2021.

[3] N. Fujieda, "On the feasibility of TERO-based true random number generator on Xilinx FPGAs," in *30th International Conference on Field Programmable Logic and Applications*, 2020, pp. 103–108.

[4] N. Fujieda and S. Takashima, "An MMCM-based high-speed true random number generator for Xilinx FPGA," *International Journal of Networking and Computing*, vol. 11, no. 2, pp. 154–171, 2021.

[5] L. Gantel *et al.*, "A FPGA-Based Post-Processing and Validation Platform for Random Number Generators," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2020, pp. 123–126.

[6] H. M. Gonzalez *et al.*, "Dynamic control of entropy and power consumption in TRNGs for IoT applications," *IEICE Electronics Express*, vol. 15, no. 2, pp. 20 171 157:1–20 171 157:11, 2018.

[7] H. Hata and S. Ichikawa, "FPGA implementation of metastability-based true random number generator," *IEICE Transactions on Information & Systems*, vol. E95-D, no. 2, pp. 426–436, 2012.

[8] W. Killmann and W. Schindler, *A proposal for: Functionality classes for random number generators, version 2.0*, Federal Office for Information Security, 2011.

[9] A. T. Markettos and S. W. Moore, "The Frequency Injection Attack on Ring-Oscillator-Based True Random Number Generators," in *11th Workshop on Cryptographic Hardware and Embedded Systems*, 2009, pp. 317–331.

[10] G. Marsaglia. Diehard battery of tests of randomness (Archived). [Online]. Available: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/

[11] U. M. Maurer, "A universal statistical test for random bit generators," in *10th Annual International Cryptology Conference on Advances in Cryptology*, 1991, pp. 409–420.

[12] *FIPS PUB 140–2 Security Requirements for Cryptographic Modules*, National Institute of Standard Technology, 2001.

[13] A. Peetermans *et al.*, "A Highly-Portable True Random Number Generator based on Coherent Sampling," in *29th International Conference on Field Programmable Logic and Applications*, 2019, pp. 218–224.

[14] O. Petura *et al.*, "A survey of AIS-20/31 compliant TRNG cores suitable for FPGA devices," in *26th International Conference on Field Programmable Logic and Applications*, 2016, pp. 1–10.

[15] A. Rukhin *et al.*, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, NIST Special Publication 800–22, Rev. 1a, 2010.

[16] R. Santoro *et al.*, "Arithmetic operators for on-the-fly evaluation of TRNGs," in *SPIE Optics + Photonics 2009*, 2009, pp. 74 440S:1–74 440S:12.

[17] M. S. Turan *et al.*, *Recommendation for the Entropy Sources Used for Random Bit Generation*, NIST Special Publication 800–90B, Rev. 1, 2018.

[18] A. Vaskova *et al.*, "Accelerating secure circuit design with hardware implementation of Diehard Battery of tests of randomness," in *IEEE 17th International On-Line Testing Symposium*, 2011, pp. 179–181.

[19] Xilinx Inc., *Vitis High-Level Synthesis User Guide*, UG1399 (v2021.1), 2021.

[20] B. Yang *et al.*, "On-the-fly tests for non-ideal true random number generators," in *2015 IEEE International Symposium on Circuits and Systems*, 2015, pp. 2017–2020.

[21] K. Yang *et al.*, "An all-digital edge racing true random number generator robust against PVT variations," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1022–1031, 2016.