Embedded HW/SW Platform for On-the-Fly Testing of True Random Number Generators

Bohan Yang*, Vladimir Rožić*, Nele Mentens*, Wim Dehaene† and Ingrid Verbauwhede*
* ESAT/COSIC and iMinds, KU Leuven,
† ESAT/MICAS, KU Leuven and IMEC,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
Email:{bohan.yang, vladimir.rozic, nele.mentens, wim.dehaene, ingrid.verbauwhede}@esat.kuleuven.be

***Abstract*—We present a HW/SW platform for on-the-fly detection of failures and weaknesses in entropy sources. By splitting the operations between hardware and software, we achieve sufficient flexibility to control the level of significance of the tests. This approach also enables sharing resources between different tests thereby reducing the area and power. Statistical tests were selected from the NIST test suite. We propose several versions of hardware co-processors for monitoring random bit sequences, ranging from 52 slices (5 tests) to 552 slices (9 tests) on Spartan-6 FPGA. We are the first to provide implementations of the Serial test and the Approximate entropy test for on-the-fly monitoring.**

I. Introduction

Random numbers are used in cryptography for generating session keys, nonces, and random challenges in various authentication protocols. These numbers are generated using specialized primitives called true random number generators (TRNGs) which produce unpredictable, uniformly distributed output values.

BSI's standard AIS-31 [1] for TRNG evaluation, requires on-the-fly tests of all TRNGs implemented in hardware. The purpose of these tests is to detect failures or statistical weaknesses of the entropy source. The draft of the special publication by NIST [2], also requires on-the-fly tests (health tests) for random number generators.

A. Previous Work

Several batteries of tests for statistical evaluation of TRNGs are available, such as FIPS [3], [4], NIST [5] and DIEHARD [6]. Some of these tests have been implemented in hardware. Hardware implementations of 4 FIPS tests are proposed in [7], [8]. Implementations of 2 simple tests from NIST and 4 from DIEHARD suites are presented in [9] and [10]. In [11], an FPGA implementation based on dynamic reconfiguration was provided. However, the architecture of this design was not suitable for on-the-fly testing. Partially reconfigurable ASIC implementations of 6 NIST tests are presented in [12]. FPGA implementations of 8 different tests from the NIST test suite are presented in [13]. Each test was implemented individually and none of the hardware resources were shared.

As pointed out in [14], one disadvantage of the embedded on-the-fly tests is the possible effect of the tests on the TRNG behavior. Embedded tests increase the chip activity resulting in more digital noise, which may affect the TRNG in such way to pass the statistical tests. However, if the TRNG is used when tests are not active, less (deterministic) noise is present on the chip. One way to resolve this situation is keeping the tests active all the time while the TRNG is in operation, thereby ensuring that the TRNG always operates under the same conditions for which it was tested. Another disadvantage is vulnerability to fault attacks. In all previous implementations, the embedded tests generate an alarm signal in case of a threat detection. If this signal is connected to ground (for instance due to a probing attack), the failure will not be detected. In order to prevent this type of fault, a different approach is required.

B. Our Contribution

In this paper we present hardware blocks for performing on-the-fly evaluation of randomness. We provide three different versions that operate on bit sequences of different lengths. The random sequence is read bit by bit. The tests were selected from the NIST battery of tests. This test suite is not designed for on-the-fly testing and in general, it is not suitable for this purpose due to the high latency and high computational requirements. However, some of these tests can be optimized for compact hardware implementation and low latency. We have selected those tests from the suite that can be optimized for compact implementation in order to make them suitable for on-the-fly monitoring.

Our first contribution is splitting the calculations between hardware and software in order to enable efficient sharing of resources between different tests in order to minimize the required hardware area. Each test is carried out using two types of operations. The first type consists of operations on the incoming bits such as: counting ones and zeros, finding the maximal longest run of the same value, counting the appearance of a given pattern or keeping track of a random walk. These operations are implemented in hardware using counters, comparators and registers, and they are performed while the TRNG is active.

The second type of operations is used for verifying the randomness hypothesis. These operations, which include addi-

This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007).In addition, this work is supported in part by the Flemish Government through FWO G.0550.12N, G.0130.13N and FWO G.0876.14N, the Hercules Foundation AKUL/11/19, and by the European Commission through the ICT programme under contract FP7-ICT-2011-284833 PUFFIN and by a gift from Intel. In addition, this work was supported in part by the Scholarship from China Scholarship Council (No.201206210295).

TABLE I: The NIST test suite. Some tests are suitable for HW implementation.

TEST	HW
1.The Frequency (Monobit) Test	Yes
2.Frequency Test within a Block	Yes
3.The Runs Test	Yes
4.Test for Longest-Run-of-Ones in a Block	Yes
5.The Binary Matrix Rank Test	No
6.The Discrete Fourier Transform (Spectral) Test	No
7.The Non-overlapping Template Matching Test	Yes
8.The Overlapping Template Matching Test	Yes
9.Maurer's "Universal Statistical" Test	No
10.The Linear Complexity Test	No
11.The Serial Test	Yes
12.The Approximate Entropy Test	Yes
13.The Cumulative Sums (Cusums) Test	Yes
14.The Random Excursions Test	No
15.The Random Excursions Variant Test	No

tion, multiplication, squaring and comparison, are performed on the obtained counters values after the sequence have been generated. These operations are only required once in a while, therefore it makes sense to reduce the datapath area at the price of increased latency, for example by sharing the multiplier between different tests. Moreover, since these operations are very common (multiplication, addition, comparison) most microcontrollers and soft-core processors already have dedicated instructions for these operations. Our assumption is that the TRNG and tests operate in an embedded system which also contains a microcontroller or a simple processor for performing basic arithmetic operations. It is also assumed that part of the cycle budget can be used for the randomness testing since this procedure is not required very often and doesn't take many cycles to perform. For this reason, operations for verifying the randomness hypothesis, are moved to the software. This is somewhat different from the standard approach where the hardware blocks complete the full test and report the failure by activating the alarm signal. Our approach assumes that the microcontroller reads the counter values from the hardware blocks and performs the remaining operations. This approach makes probing attacks difficult because there is no single alarm signal, but rather a set of numerical values that are being transmitted.

Another advantage of HW/SW co-design is flexibility. Each test can be carried out with a critical value α of level of significance, where the recommendations by NIST are that α is chosen from the interval $[0.001, 0.01]$. The presented hardware blocks analyze the generated sequence and provide the results that do not depend on α. Level of significance only figures in the operations performed by the software which can be easily programmed.

Our second contribution is the implementation of the serial test and the approximate entropy test from the NIST test suite. To the best of our knowledge, these are the first hardware implementations of these tests suitable for on-the-fly testing.

Our third contribution is the unified implementation of hardware blocks, which allows for different tests to share resources such as bit counters. In addition, different tests use the same counter values (for example, number of ones in a sequence) so there is no need for these counters to be duplicated.

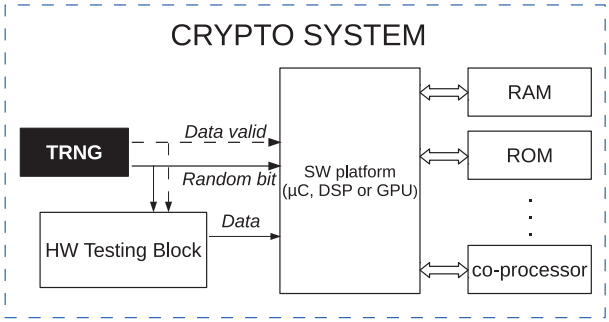

Fig. 1: Testing environment

C. Organization

This paper is organized as follows. In Section 2, we provide the general background and notation. Section 3, deals with the implementation aspects including the numerical simplifications, splitting the operations between the hardware and software as well as the implementation of the hardware blocks. The results of the FPGA and ASIC implementations, as well as comparison with relevant work are presented in Section 4. Finally, conclusions and proposals for future work are provided in Section 5.

II. Background

A. Statistical Tests

Even though randomness is a property of a variable, rather than a sequence of numbers, and therefore cannot be measured like other physical quantities, it is possible to estimate the randomness by checking different statistical properties. The idea behind statistical tests is to start with the hypothesis that the RNG is ideal, we will denote this hypothesis as H_0. A statistical test is used to measure a property of the generated sequence, for example the longest run of zeros. Based on the test result, the hypothesis H_0 is either accepted or rejected. If the sequence is indeed random, it is still possible to reject H_0 with some small probability. This is known as a type 1 error. The probability of this error is a design parameter called the level of significance and denoted as α. The NIST recommendation for the value of α is between 0.001 and 0.01. The other type of error that can occur is accepting H_0 when the sequence is not random. This is known as a type 2 error, and the purpose of the test is to minimize the probability of this error. The generated sequence should be tested for many different statistical weaknesses in order to accept it as random. As summed up in table I, The NIST suite consists of 15 statistical tests which estimate different properties of a random variable. These tests are originally designed to evaluate the statistical properties of pseudo-random number generators (PRNGs), but they are also used for evaluation of TRNGs. Hardware implementations of these tests can be used for on-the-fly monitoring.

B. On-the-fly Tests

Hardware implementations of TRNGs are susceptible to different active attacks. It is possible to reduce the randomness by changing the operating conditions, such as temperature or

TABLE II: Calculations split between hardware and software.

Test	Hardware	Software
Frequency Monobit Test	N_{ones}	Comparison operations
Frequency Test Within a Block	$\varepsilon_1, \varepsilon_2 ... \varepsilon_N$	$\sum_{i=0}^{N}(\varepsilon_i - \frac{M}{2})^2$
Runs Test	N_{ones}, N_{runs}	Comparison operations
Longest Run of Ones in a Block	$\nu_{runs,1}, \nu_{runs,2} ... \nu_{runs,N}$	$\sum_{i=0}^{N} \nu_{runs,i}^2 (\frac{1}{\pi_i})$
Non-overlapping Template Matching Test	$W_1, W_2 ... W_N$	$\sum_{i=1}^{N}(2^m W_i - \mu 2^m)^2$
Overlapping Template Matching Test	$\nu_{temp0} ... \nu_{tempN}$	$\sum_{i=0}^{N} \nu_{temp,i}^2 (\frac{1}{\pi_i})$
Serial Test	$\nu_{0000}, \nu_{0001}, ... \nu_{1111}$, $\nu_{000}, \nu_{001}, \nu_{010} ... \nu_{111}$, $\nu_{00}, \nu_{01}, \nu_{10}, \nu_{11}$	$\Psi_m^2 = \frac{2^m}{n} \sum_{i_1 ... i_m} \nu_{i_1 ... i_m} - n$ $\Psi_{m-i}^2 = \frac{2^{m-1}}{n} \sum_{i_1 ... i_{m-1}} \nu_{i_1 ... i_{m-1}} - n$ $\Psi_{m-2}^2 = \frac{2^{m-2}}{n} \sum_{i_1 ... i_{m-2}} \nu_{i_1 ... i_{m-2}} - n$ $\nabla \Psi_m^2 = \Psi_m^2 - \Psi_{m-1}^2$ $\nabla^2 \Psi_m^2 = \Psi_m^2 - 2\Psi_{m-1}^2 + \Psi_2^2$
Approximate Entropy Test	$\nu_{0000}, \nu_{0001}, ... \nu_{1111}$, $\nu_{000}, \nu_{001}, \nu_{010} ... \nu_{111}$	$\sum_{i(m=3)} \frac{\nu_i}{n} log \frac{\nu_i}{n} - \sum_{i(m=4)} \frac{\nu_i}{n} log \frac{\nu_i}{n}$
Cumulative Sums Test	$S_{max}, S_{min}, S_{final}$	$max(S_{final} - S_{min}, S_{max} - S_{final})$ Comparison operations

voltage. Paper [15] demonstrates that manipulating the power supply can cause ring oscillators inside a TRNG to lock to a certain frequency, thereby reducing the generated entropy. A similar attack can be done using an electromagnetic probe as shown in [16]. The trivial way for completely disabling the source of randomness is by cutting the signal wire used for transmitting random bits. In addition to active attacks, a designer needs to worry about failures due to aging. For this reason, different tests are required for on-the-fly monitoring of RNGs: quick tests for fast detection of the total failure of the entropy source, as well as slow tests for the detection of long term statistical weaknesses.

III. Implementation

A. System Design

In embedded systems, random number generators are never used as a stand-alone module, but rather in conjunction with the other components such as embedded processors, microcontrollers or DSPs. For this reason, we can assume that the chip containing an RNG and the HW testing block, also contains a component that can perform basic arithmetic operations. This component can serve as the software platform.

As depicted in Fig. 1, the embedded system consists of a TRNG, the HW Testing Block, at least one component that performs arithmetic (microcontroller, DSP or a GPU), and possibly other components such as embedded RAM and crypto co-processors. We split the testing implementation into two parts: hardware and software. To reduce the area and power consumption of the HW testing block, it is implemented in a compact manner using only the basic components such as counters, comparators and registers, while all power-hungry arithmetic operations are moved to the software part. Squaring, multiplication, logarithm and comparison with the precomputed constants are performed in software.

The proposed approach allows for a flexibility with respect to the level of significance α. Since the implementation of the hardware block doesn't depend on α, the software part can be updated in case this value needs to be changed. As pointed out in [14], on-the-fly tests should be active while the TRNG is working in order to ensure that they are always operating in the same conditions as when they are being tested. The proposed approach allows us to run the hardware block all the time and check the test results only when needed.

B. HW/SW Calculations

All calculations needed for the statistical tests, are divided between a HW testing block and software executed on the SW platform (micro-controller or any processor with instructions for basic arithmetic). The boundary between hardware and software is chosen to minimize the hardware block, i.e. to keep only the necessary parts in hardware. Each test consists of operations that have to be executed while the bit stream is generated (the HW part) and basic arithmetic operations (SW part). It is also important to minimize the amount of data that needs to be transferred from HW to a co-processor in order to simplify the interface between the two modules.

We have selected 9 tests from the NIST test suite which are suitable for this type of implementation, as indicated in table I. The remaining 6 tests either require too much data storage in the HW module which would result in large area, too complex operations in the software part which would result in high latency, or too much data to be transferred between the two modules which would result in an overly complicated interface.

Table II presents how the required operations were divided between HW and SW. The middle column (Hardware), lists all the values that are computed by the HW module and transferred to the co-processor. We used the following notation:

- N_{ones} - the total number of ones.

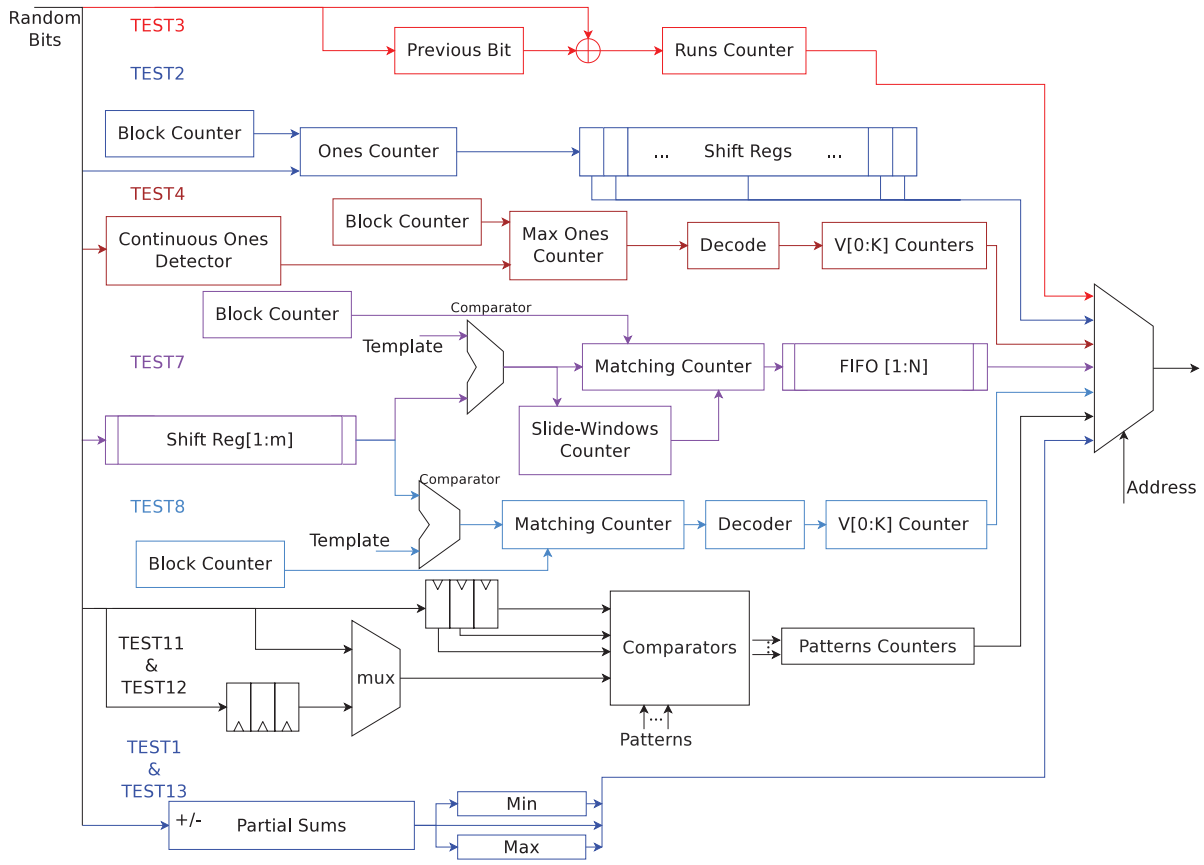

Fig. 2: Hardware module containing all tests

- N_{runs} - the total number of runs in a sequence. A single run is a consecutive appearance of a single value (0 or 1).
- ε_i - number of ones within a block of data.
- M - length of a single block of data.
- N - number of data blocks.
- ν_i - number of runs within a block of data.
- W_i - number of non-overlapping appearances of a given template within a block of data.
- $\nu_{temp,i}$ - the number of data blocks in each category depending on the number of overlapping appearances of a given template.
- m - length of a template.
- $S_{max}, S_{min}, S_{final}$ - partial sums obtained by the up/down counter. Maximal, minimal (negative) and the final value are recorded.

All other values are test-specific precomputed constants.

Software routines operate on these obtained data values. As can be seen from the last column of table II, required operations for software are basic multiplication, addition and comparison operations. The only difficulty is implementing the $x \cdot log(x)$ function needed for the approximate entropy test, which will be described in more detail in subsection III-D.

C. HW Implementations

We have implemented several versions of the HW testing block. Fig. 2 shows the implementation of the largest version that contains all 9 tests and operates on a sequence of 2^{20} bits. Clock and enable signals are omitted for better clarity. The global bit counter is also not shown. This counter is used to count the total number of bits in order to detect the end of the sequence. A memory-mapped interface is implemented using a large multiplexer, where the 7-bit address is used as a select signal. Since this interface contributes significantly to the overall area we can save resources by reducing the number of transmitted values.

After receiving each random bit from the generator, all update calculations finish within one clock cycle.

This type of unified implementation enables us to share more resources between different tests to obtain higher area reduction. We have used 4 tricks to reduce the area of this module:

- **Omitting a redundant counter:** Tests 1 and 3 use the total number of 1's in a sequence to compute

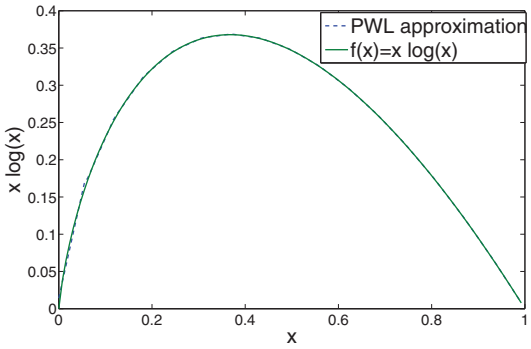

Fig. 3: PWL approximation of the function $x \cdot log(x)$

the test result. However, it is possible to obtain this result without this counter. For the implementation of test 13, an up/down counter is used to keep track of the random walk. The total number of ones can be calculated from the final value of this counter. For this reason, the counter of ones can be omitted.

• **Block detection:** For the implementations of tests 2, 4, 7 and 8 it is necessary to divide the sequence into sub-blocks and look for certain properties in each block (the total number of ones, longest run of the same bit value, and occurrences of different patterns). We have selected test parameters such that block lengths are equal to powers of 2, which enables us to detect the beginning and the end of each block by simply observing specific bits of the global bit counter.

• **Unified implementation:** The approximate entropy test (test 12) uses the number of all 4-bit and 3-bit patterns in a sequence. These values are already provided by the serial test (test 11) implementation, therefore there is no need for the separate implementation of test 12.

• **Shared shift register:** The non-overlapping and the overlapping template match tests compare the generated numbers with the pre-defined 9-bit patterns. The same shift register can be used for both tests.

D. SW Implementations

Typical software implementations of statistical tests operate by computing the P-value, and comparing it with the required level of significance. P-value is the probability that an ideal random number generator produces a sequence which is worse than the measured sequence with respect to the metric used by the test (for example bias, or the longest run of the same bit value). This is a computationally intensive task because calculating P-values requires complicated functions such as *erfc* and the *gamma* function. We use a simple approach of computing the inverse functions of the critical value and storing the precomputed constants, thereby skipping the most computationally intensive step. This approach is also used in [9], [13], [12].

As shown in table II, implementations of tests 1 and 3 only require comparison operations. Test 1 only compares the

TABLE III: Implementation results

	n=128		n=65536			n=1048576		
	light	medium	light	medium	high	light	medium	high
test1	•	•	•	•	•	•	•	•
test2	•	•	•	•	•	•	•	•
test3	•	•	•	•	•	•	•	•
test4	•	•	•	•	•	•	•	•
test7				•	•		•	•
test8							•	•
test11		•			•			•
test12		•			•			•
test13	•	•	•	•	•	•	•	•
FPGA:								
Slice	52 0.8%	149 2.2%	144 2.1%	168 2.5%	377 5.5%	173 2.5%	291 4.3%	552 8.1%
FF	110	329	307	375	836	379	585	1156
LUTs	158	471	420	454	1103	546	828	1699
MaxFreq (MHz)	156	147	143	136	133	125	122	121
ASIC:								
GE	1210	3632	3243	3850	8983	4013	5993	12416
SW:	16-bit instructions							
ADD	9	153	108	122	266	130	358	890
SUB	8	14	16	24	30	24	40	50
MUL	4	28	24	24	48	15	47	91
SQR	8	36	14	22	50	23	45	101
SHIFT	0	3	0	8	11	0	8	11
COMP	22	28	42	44	50	34	42	48
LUT	0	24	0	0	24	0	0	24
READ	10	24	18	22	50	21	35	91

N_{ones} with the critical value. For test 3, the critical values for the N_{runs} are stored in the program memory as constants and they depend on the N_{ones}. The SW procedure first checks the interval where N_{ones} belong and based on the result, compares N_{runs} with the appropriate constant. Similar approach was used for FPGA implementation in [13].

Other tests require simple calculations on the obtained values before the comparison. Required operations are comparison, addition (subtraction), multiplication and squaring. Typical processor has dedicated instructions for these operations. The main difficulty is related to the implementation of The Approximate Entropy Test, which required the implementation of the function $x \cdot log(x)$. In order to avoid computationally intensive logarithm calculation, we implemented this function using piece-wise linear approximations with 32 segments. As can be seen on Fig. 3, the approximation (dash line) is almost indistinguishable from the function (full line) resulting in less than 3% error.

IV. Results

We made 8 different designs which covered 9 tests from the NIST test suite. We implement our hardware designs in Verilog HDL and use Mentor Graphics Modelsim SE PLUS 6.6d for functional simulation. All proposed hardware designs are synthesized using Xilinx ISE14.7 on Spartan-6 XC6SLX45 FPGA and Synopsys Design Compiler D-2010.03-SP4 to UMC's $0.13\mu m$.1P8M Low Leakage Standard cell Library with typical values (voltage of $1.2V$ and temperature of $25\,^{\circ}\mathrm{C}$).

As with most practical implementations, there is no golden way to the perfect system in a generic way, and different applications demand different design trade-offs. As shown in table III, we propose 8 different implementations which support three different input lengths $128/65536/1048576$ bits.

TABLE IV: Comparison

	[13]	This work
	Sequence length	
test1	20000	65536
test2	20000	65536
test3	20000	65536
test4	128	65536
test7	2048	65536
test13	20000	65536
Slices	256	168
Latency	21	4909

For hardware design, with the merit of a compact hardware footprint, the 128-bit version can be utilized for lightweight designs for up to seven different tests. On the other hand, the 1048576-bit version has the capability to support long term evaluation and up to nine different tests. The 65536-bit version provides a balanced trade-off between the hardware area and the input length of the random sequence. All our implementations on FPGA have a maximum working frequency larger than $100MHz$, in other words, they can handle an input bit rate of $100Mbit/s$, which is enough for most of the TRNGs on FPGA.

Our designs are also suitable for ASIC, either as an individual unit or as a building block for processors. In table III we provide the area results in GE (gate equivalence).

Our software designs can be implemented on different hardware platforms, such as microcontroller, GPU and DSP. These embedded systems might utilize different dedicated peripherals, such as a HW multiplier and squarer. The number of required clock cycles is greatly dependent on the SW platform. In table III, We present the instruction count of software implementations for a 16-bit architecture. We can see that the largest version requires more than 900 ADD/SUB and almost 200 MUL/SQR. The reason is that instructions operating on data larger than 16-bit have to be decomposed into several 16-bit operations. We can expect that, on 32-bit or 64-bit platforms, considerably lower latency could be achieved.

Since this is the first unified FPGA implementation of the embedded tests, it is difficult to compare with previously published work. In [13], individual implementations of different tests are presented. These tests are operating on sequences of different lengths, as shown in table IV. By comparing the total number of occupied slices of the individual tests from [13] with our unified implementation for a sequence length of 65536 bits, we can see that our implementation uses around 20% less slices. However, the comparison is not entirely fair for two reasons: one, because we use a longer bit sequence, and two, because some of the functionality is moved to software. In order to compare the latency, we utilize openMSP430 [17] as the hardware platform to evaluate our design. As expected, the latency of the software routine is higher than the latency of the slowest test from [13] but still much lower than the time needed to generate the sequence.

V. Conclusions and Future Work

In this paper, we have presented a unified implementation of different NIST tests based on splitting the operations between hardware and software. By keeping only the necessary operations in hardware, we have achieved our goal of a low area cost of the hardware part and a sufficient flexibility for the software part.

There are several topics to explore as part of the future work. One topic is modifying the hardware blocks to allow for more flexibility, for example by allowing the software to select the length of the test sequence, as well as the test parameters. Another topic for future research is implementing the remaining tests from the NIST test suite as well as testing the software implementations on different types of microcontrollers and open-core processors.

References

[1] W. Killmann and W. Schindler, "A proposal for: Functionality classes for random number generators," ser. BDI, Bonn, 2011.

[2] E. Barker and J. Kelsey, "Recommendation for the entropy sources used for random bitgeneration," ser. NIST DRAFT Special Publication 800-90B, 2012.

[3] "FIPS 140-1, Security requirements for cryptographic modules," 1999.

[4] "FIPS 140-2, Security requirements for cryptographic modules," 1999.

[5] A. Rukhin et al., "A statistical test suite for random and pseudorandom number generators for cryptographic applications." Special-Pub:800-22 NIST, August 2008.

[6] "DIEHARD battery of tests of randomness."

[7] R. Santoro, O. Sentieys, and S. Roy, "On-line monitoring of random number generators for embedded security." in *ISCAS*. IEEE, 2009, pp. 3050–3053.

[8] ——, "On-the-Fly Evaluation of FPGA-Based True Random Number Generator," in *IEEE Computer Society Annual Symposium on VLSI, ISVLSI'09*, May 2009.

[9] A. Vaskova, C. López-Ongil, A. Jiménez-Horas, E. San Millán, and L. Entrena, "Robust cryptographic ciphers with on-line statistical properties validation," in *IOLTS*, july 2010, pp. 208 –210.

[10] A. Vaskova, C. López-Ongil, E. San Millán, A. Jiménez-Horas, and L. Entrena, "Accelerating secure circuit design with hardware implementation of diehard battery of tests of randomness," in *IOLTS*, july 2011.

[11] D. Hoţoleanu, O. Creţ, A. Suciu, T. Gyorfi, and L. Văcariu, "Real-time testing of true random number generators through dynamic reconfiguration," in *DSD*, sept. 2010, pp. 247 –250.

[12] V. B. Suresh, D. Antonioli, and W. P. Burleson, "On-chip lightweight implementation of reduced nist randomness test suite," in *HOST 2013*. IEEE, 2013, pp. 93–98.

[13] F. Veljković, V. Rožić, and I. Verbauwhede, "Low-cost implementations of on-the-fly tests for random number generators," in *DATE 2012*. IEEE, 2012, pp. 959–964.

[14] V. Fischer, "A Closer Look at Security in Random Number Generators Design," in *COSADE*, 2012, pp. 167–182.

[15] A. T. Markettos and S. W. Moore, "The frequency injection attack on ring-oscillator-based true random number generators," in *CHES*, 2009, pp. 317–331.

[16] P. Bayon, L. Bossuet, A. Aubert, V. Fischer, F. Poucheret, B. Robisson, and P. Maurine, "Contactless electromagnetic active attack on ring oscillator based true random number generator," in *COSADE 2012*, ser. LNCS, vol. 7275, 2012, pp. 151–166.

[17] O. Girard, "Openmsp430 project," *available at opencore. org*, 2010.