Computing and Informatics, Vol. 32, 2013, 1272–1292

IMPLEMENTATION OF A RANLUX BASED PSEUDO-RANDOM NUMBER GENERATOR IN FPGA USING VHDL AND IMPULSE C

Agnieszka Dąbrowska-Boruch, Grzegorz Gancarczyk
Kazimierz Wiatr

AGH University of Science and Technology
Department of Electronics
al. A. Mickiewicza 30
30-059 Kraków, Poland
&
ACC CYFRONET AGH
ul. Nawojki 11
30-950 Kraków, Poland
e-mail: {adabrow, gegula, wiatr}@agh.edu.pl

Abstract. Monte Carlo simulations are widely used e.g. in the field of physics and molecular modelling. The main role played in these is by the high performance random number generators, such as RANLUX or MERSSENE TWISTER. In this paper the authors introduce the world's first implementation of the RANLUX algorithm on an FPGA platform for high performance computing purposes. A significant speed-up of one generator instance over 60 times, compared with a graphic card based solution, can be noticed. Comparisons with concurrent solutions were made and are also presented. The proposed solution has an extremely low power demand, consuming less than 2.5 Watts per RANLUX core, which makes it perfect for use in environment friendly and energy-efficient supercomputing solutions and embedded systems.

Keywords: RANLUX, FPGA, PRNG, HPC, HPRC, VHDL, Impulse C

Mathematics Subject Classification 2010: 68U01, 65P20, 60G099

1 INTRODUCTION

The ACC Cyfronet AGH is the biggest high performance computing centre in Poland [17]. The main motivation for this work was to ease CYFRONET scientific users with their complex computational and time consuming tasks. The main problem for the users focused on Astrophysical Competence Centre of CYFRONET are the extremely time consuming Monte Carlo (MC) simulations, used in e.g. the SuperB experiment [18]. In the SuperB experiment, the MC simulations last longer than any other calculations taken and consume more than 670 kHEP-SPEC06 of the HPC centres computational power per year [19].

As for the MC experiments themselves, they rely on random sampling, therefore the fast and reliable generation of (pseudo-) random numbers is extremely important. Among others the Organisation Européenne pour la Recherche Nucléaire (CERN) recommends only three PRNGs (Pseudo-Random Number Generators) for MC simulation purposes, i.e. the RANMAR, the RANECU and the RANLUX implementations [8]. These PRNGs are used i.a. in the following packages: FermiQCD, UKQCD, SZIN, ISAJet, GEANT4, PYTHIA, HERWIG, SHERPA, ALPGEN. The MERSENNE TWISTER is also being frequently used in the MC applications, as one with an extremely long period and satisfying the randomness of results [8]. As for the CYFRONET users dealing with High Energy Physics (HEP), only the RANLUX was in their scene of interest.

The authors goal was to check, if implemented in the hardware the RANLUX core suits as an external accelerator of the PRNs generation process. Special care was taken to investigate, how such a solution could be attractive for super computers centres.

The measurements show that the single RANLUX algorithm instance run on a 5th generation FPGA device (currently the 7th generation is being brought into the market) is consuming less than 2.5 W with 40 % of the performance of Intel's low power processor with 35 W of Total Dissipated Power (TDP), and 45 % of the performance of an AMD high performance processor with 125 W of TDP.

In Section 2 there is a brief background of the RANLUX algorithm, its history and solutions currently used. Section 3 is fully devoted to a description of the RANLUX algorithm implementations in the VHDL and Impulse C languages, a few words about the languages used and the reconfigurable platform, which was exploited during the tests can be found there as well. Section 4 contains comparisons of all the aforementioned solutions, while Section 5 gives information about present research and future plans of development. At the end of the article Appendixes A and B can be found. In both implementations of the RANLUX algorithm cores done in Impulse C (A) and VHDL (B) were inserted.

2 BACKGROUND

The RANLUX algorithm is a pseudo-random number generator. It is one of the standard generators used for the Monte Carlo simulations. The RANLUX was developed by Martin Lüscher in 1994 [1]. The author of the RANLUX PRNG started his work with the existing SWB (Subtract-With-Borrow) RCARRY algorithm proposed by Marsaglia and Zaman [2]. The RCARRY PRNG has a generation period in the order of 10^{171}. At first it seemed to have very good statistical properties, but now it is well known that the RCARRY fails some of the newer randomness tests. This fact was Lüscher's motivation to create a new algorithm for the pseudo-random number generator. The main assumption was that a new PRNG should pass all known statistical tests.

2.1 The RANLUX Algorithm

The RANLUX algorithm was developed using the theorem of the Kolmogorov entropy and the Lyapunov exponent. A single RANLUX algorithm step needs to perform two stages for computation of a single precision floating point or integer PRN. The first stage of this single step is the determination of the Δ_n value from Equation (1) [7]

$$\Delta_n = x_{n-s} - x_{n-r} - c_{n-1}, \tag{1}$$

where $s = 10$, $r = 24$ in the original RANLUX algorithm.

The Δ_n value is calculated for $n \geq r$. It means that the algorithm starts with $n = 24$. The next stage of the single algorithm's step is to determine the values of the x_n and the carry c_n. These values are calculated from a dependence (2)

$$x_n = \begin{cases} \Delta_n, & c_n = 0 \quad \text{if} \quad \Delta_n \geq 0 \\ \Delta_n + m, & c_n = 1 \quad \text{if} \quad \Delta_n < 0 \end{cases}, \tag{2}$$

where $m = 2^{24}$ and x_n is an integer in the range $0, \ldots, 2^{24} - 1$.

In the basic specification of the RANLUX, the output value can be integer x_n or floating point $x_n/2^{24}$ in the range $0, \ldots, 1$.

The basic core of the RANLUX algorithm needs the first 24 values $x_0, \ldots, x_{23}$ and the carry bit c_{23} to function properly. In other words the RANLUX algorithm needs a total of 576 bits to start-up. These bits can be generated using Equation (3) [7]

$$b_n = (b_{n-13} + b_{n-31}) \bmod 2, \tag{3}$$

where b_n indicates the individual bits and $n \geq 0$.

The dependence (3) requires 31 initial bits for a proper start of recursion. These initial bits can be represented by an integer seed with a value between 1 and $2^{31} - 1$.

The author of the RANLUX algorithm noticed that omitting a few generated numbers from Equation (2) improves statistical properties of the PRNG. It means that the PRNG's sequence of numbers has better randomness than the simple RCARRY algorithm. Lüscher introduced so-called luxury levels for this reason. The RANLUX algorithm has five luxury levels: 0–4. The basic rule for the RANLUX algorithm is that r generated pseudo-random numbers are delivered, while

$p-r$ generated numbers are omitted in every pack of p generated PRNs. All p numbers (both r used and $p-r$ omitted) are used for the further recursive calculations. Lüscher has determined the r value as 24. The exact values for every luxury level were given in Table 1.

Luxury level	Delivered numbers (r)	Omitted numbers ($p-r$)	All generated numbers in one algorithm pass (p)
0	24	0	24
1	24	24	48
2	24	73	97
3	24	199	223
4	24	365	389

Table 1. The luxury levels of the RANLUX algorithm [1]

The usage of luxury level 4 ensures that the sequence of delivered numbers has a chaotic character. Additionally, luxury level 4 guarantees preservation of the generator's period [4]. A further increasing of numbers of the omitted values does not make sense, because it does not improve the randomness of the generated sequence, but it requires much more calculation effort per each 24 random numbers in a generated sequence.

The most popular is luxury level 3. It is satisfying from the statistical properties point of view. As for luxury level 2, the random number sequence can have defects in theory (low range correlation between samples), but in practice no defects have been detected yet [4].

Please note that the levels with the best statistical properties use prime p parameter values and only for those the RANLUX algorithm has passed one of the most important statistical property tests, the DIE HARD battery of tests of randomness and crush.

2.2 Existing Implementations

Solutions currently used for PRNs (Pseudo-Random Numbers) generation due to the RANLUX's guidelines are the ones for the General Purpose Processors (GPPs), i.e. those based on x86, Cell, Spark, ARM, etc. architecture and for General Purpose Graphic Processing Units (GPGPUs), i.e. AMD's (former ATI) or nVidia's graphic cards.

The graphic card should be considered as an external accelerator (connected with the GPP by the PCI-E bus), its role is to generate only random numbers and send them to the GPP for further processing, or as autonomic computational systems, which are meant to generate PRNs and make additional processing locally (i.e. on the card) with no GPP usage. In this second case the role of the GPP is reduced to run only the operational system and send initial data to the graphic card.

In this paper graphic cards will be considered as external accelerators only and not as whole, independent, computational systems.

The RANLUX implementations are written in the ANSI C (with Assembler inserts), C++, Fortran, Java, OpenCL, CUDA C and nVidia's ASM programming languages. The performance of the Java solutions stand out from the others, therefore they will not be discussed here, just like the solutions written in nVidia's ASM will not be taken into account, because they are currently out-of-date. As for the Fortran implementation, it was done by James [3], just as the ANSI C one done by Lüsher. Both have a similar performance.

The classical solutions run on the GPPs can use up to 4 independent threads per core (thanks to the SSE registers usage) and run on up to 6 cores independently. This gives up to 24 RANLUX instances run in parallel on one processor (e.g. Intel Xeon X5670). Each instance is the RANLUX algorithm run with different initial conditions (seeds). The produced samples are still random as long as luxury level of 3 or 4 is used. For the lower luxury level values, a correlation between the output of a single instance is visible, *ergo* a correlation between the results of two independent instances occurs even more clearly. Lüsher's ANSI C + ASM solution of RANLUX (version 3.3) is available at his homepage [20] and distributed under the GNU General Public License (GPL) conditions.

The C++ SISCone (Seedless Infrared Safe Cone) implementation uses only one core and one thread per core in its original form. It is used as a part of the larger HEP project simulation, where MC experiments are involved. The code and more information about the SISCone jet algorithm are available at [21] and distributed also under the GNU GPL conditions.

For both CUDA C and OpenCL solutions presented in [8] and [9], code is not accessible online. As for the RANLUX's OpenCL implementation, large parts of used code were inserted in the Demchick's paper [8].

Because all four implementations differ from the original one given by Lüscher, the authors decided to:

- modify the ANSI C + ASM and C++ code to have the same p parameter value as the one given by Lüscher originally,
- run GPP solutions in one core, one thread mode only, which ensures the same measuring conditions as the ones for FPGAs,
- cite results just as they were in the case of the OpenCL and CUDA C implementations.

One of the first publications with the results of the software implementation of the RANLUX PRNG is the Hamilton and James [5] article. They implemented the original Lüscher's algorithm with luxury levels 0–4. The implementation was made using the Fortran language.

An estimation of single RANLUX instance efficiency was done using the efficiency parameter, which uses a kSa/MHz unit, due to a well known approximation (4)

$$P \sim f, \tag{4}$$

which denotes that the power P consumed by the working processing unit is proportional to its clock speed f. Equation (4) is a generalised version of more complex

$$P = CV^2 f, \tag{5}$$

which can be found i.a. in [24] and [25]. C denotes total capacitance of the processing unit, while V is its supply voltage. The CV^2 factor for all considered hardware platforms differs by less than 15 %, therefore it was simply neglected.

Demchik in [8] presented the implementation of the RANLUX algorithm with different luxury levels (Table 2) than the ones in the original Lüscher's algorithm. Luxury level 1 is the same as in the original RANLUX, but the next levels are different. In Demchik's solution: 96 numbers are omitted from luxury level 2, 216 numbers are omitted from luxury level 3 and 384 numbers are omitted from luxury level 4. All these numbers are multiples of 24. Demchik wrote that the GPU had shown better performance for such defined luxury levels. He archived method 1.11 GSa/s (random numbers/samples per second) in the global buffer at luxury level 3, but for 4 096 RANLUX threads with different seed values. It means that one thread produces only 271 kSa/s. It also denotes, that this solution has a total efficiency of 1.53 MSa/MHz at luxury level 3 (the ATI Radeon HD 5 850 GPGPU core works with a 725 MHz clock speed [8]) and 0.37 kSa/MHz of efficiency per single GPGPU thread. The author has also implemented a so-called planar method and achieved 10.85 GSa/s at luxury level 3. There is no information in [8] how many RANLUX instances with different seed values were used in this solution.

Another GPU implementation of the RANLUX algorithm is Wende's work [22]. In this work the luxury levels are also different from the Lüscher's algorithm. Wende's luxury levels are shown in Table 2. The implementation was done using an nVidia Tesla C1060 graphic card. The author had used 240 thread blocks and each thread block contained 16 CUDA threads. This implementation had 3 840 sub-RANLUX instances. Each RANLUX instance worked with a different seed value. This solution generated 720 MSa/s at luxury level 1, i.e. 3 MSa/s per single thread block and 187.5 kSa/s per one sub-RANLUX instance. This solution stands at the level of a total efficiency of 1.20 MSa/MHz (the clock speed of the Tesla C1060 GPGPU core is equal to 602 MHz [22, 23]) and the efficiency of a single thread of 0.31 kSa/MHz at luxury level 1.

3 DESCRIPTION OF HARDWARE IMPLEMENTATIONS

The hardware-aided generation of PRNs was done using the DRC AC2020 reconfigurable system [11]. The mentioned workstation is equipped with an AMD Opteron

Luxury level	Original RANLUX [1]	RANLUX 3.0 [7]	Demchik's planar [8]	Wende [22]
0	24	133	–	109
1	48	226	24	202
2	97	421	120	397
3	223	–	240	–
4	389	–	408	–

Table 2. The p parameter values (number of PRNs generated during one algorithm pass) in different RANLUX implementations

2 352 GPP working with a frequency of 2.1 GHz. The AC2020 also has one FPGA device (a so called Reconfigurable Processing Unit – RPU) inside the motherboard's second CPU socket. Both the GPP and RPU are connected by a 64 bit wide HyperTransport v1.0 bus [14]. In the case of the AC2020, the RPU used is the Xilinx's Virtex5 LX220-2 FPGA running with a maximum frequency of 200 MHz. The RPU clock can be downgraded (e.g. to 167 MHz) if needed, but in the case of the implementations shown in this article, it was always working at the highest possible clock speed.

The GPP role was to start up (by sending a seed) the RANLUX generator implemented inside the RPU and collect the generator's output data, therefore no extra pre- or post-processing was done by the GPP.

Inside the RPU a single RANLUX instance was implemented. By "instance", a single RANLUX generator core is meant. The implemented algorithm is 100 % compatible with the RANLUX v3.0 specification [7], including the initialisation given by Equation (3) and the luxury levels as the ones shown in Table 1.

What is worth emphasising is that the initialisations other than the one given by Lüscher (Equation (3), [7]) and used by the authors of [8, 9, 22] are more GPP/GPGPU oriented than the default one and that the usage of p parameter values other than the ones given by Lüscher (Table 1, [7]) *may not* fulfil the Kolmogorov's conditions, *ergo* the whole assumption about the randomness of output data obtained by the authors of [8, 9, 22] using their own modification of the original algorithm is at least doubtful. Also none of the articles says a single word about verifying the results with the DIE HARD battery tests. As in the case of the initialisation function, the usage of the p parameter other than the default one makes the algorithm more GPP/GPGPU oriented, significantly increasing its performance.

3.1 Impulse C

Impulse C is a High Level Synthesis/Language (HLS/HLL) used to program custom applications targeting FPGA devices. It is an ANSI C based language adapted to be used in the process of reconfigurable logic designing, with additional special hardware oriented functions and C-to-HDL compiler pragmas. Both functions and pragmas give programmers an opportunity to implement their algorithms in a fully parallel manner and by adapting hardware to the algorithm, not opposite, as in the case of GPP, where hardware architecture is stiff and cannot be changed.

The Impulse C license is currently held by the Impulse Accelerated Technologies Inc. [10]. The company delivers the Impulse C language standard, the CoDeveloper IDE for programming, the simulation and verification processes, C-to-HDL compiler and also the Platform Support Packages (PSPs), i.e. special libraries designed especially for a given vendor (e.g. Xilinx, Altera), device (e.g. Virtex5, Stratix IV) or even reconfigurable platforms (e.g. DRC AC2020, Pico E-16). Such solutions not only make it easy to write its own code and implement it, but also to port any code written once to any platform, by simply choosing a different PSP.

Impulse C uses a stream oriented, process based programming model, where processes (threads) can be run both using a GPP (a software process) and RPU (a hardware process). Also, there can be many hardware and software processes running in parallel or concurrently. The processes exchange information using so called streams, which are in fact FIFOs (First In First Out registers). In some cases the processes can exchange information using other solutions (it depends on the PSP support), like shared memory banks. Nevertheless, the stream remains as the main communication path. Beside Impulse C, there are many other, similar HLSs/HLLs (e.g. Handel-C, AutoESL, Mitrion-C) [15, 13, 16], but none is as mature as Impulse C, developed and designed especially for high performance reconfigurable computing purposes. Also, only Impulse C gives an opportunity to write the whole application (both hardware and software parts) at once with one tool.

The block diagram showing the data flow for the RANLUX-based PRNG implemented in the DRC AC2020 system using the Impulse C language is shown in Figure 1.

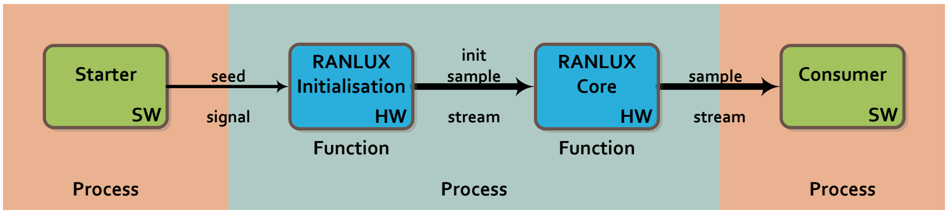

Figure 1. Data flow diagram of the RANLUX based PRNG implemented in Impulse C

The diagram shown in Figure 1 is divided into two parts composed of three processes. In the software part, two processes are working. The first one, the Starter, passes the seed to the hardware generator using a 32 bit wide signal (only 31 bits of the passed seed are used inside the generator core). The signal is an Impulse C synchronisation solution, which additionally carries valid data and can be faster than the stream, while sending a small portion of data as in this case. The RANLUX core is inactive until the starting signal triggers it.

The second software process, the Consumer, receives data from the generator and writes it into the memory, file, etc.

The communication between the generator and the Consumer is made using a 32 bits wide stream working in a burst mode. The burst mode, known for example from the USB interface, makes that a pack of data (e.g. 1 024 samples) is being transmitted, instead of a single sample. Such a solution increases the transfer rate, because the stream is opened and closed only once, instead of N times, where N is the number of samples in a single pack. The samples are transmitted through the output stream as long as the users want it to. Physically, both the stream and signal use this same medium, i.e. 64 bit wide HyperTransport v1.0 bus connecting the GPP with the RPU.

The RANLUX generator implemented inside the RPU is composed of two components (functions), where both are run inside the same process simply called ranlux. The first function, called RANLUX_Initialisation, is triggered by the input signal and generates the first 24 samples using the seed value passed by this signal. The generation is done in regard to Equation (3), but instead of obtaining one bit per clock cycle, it calculates the whole about 12 times faster (576 bits in 47 clock cycles). When the initialisation ends, the main RANLUX function is run (*RANLUX_core*). It produces one sample per clock cycle. Such a performance was obtained by pipelining the whole process and by implementing the samples memory circular buffer, needed by the generation process, in the form of a register. Registers, unlike memories, have low access time (no addressing is needed, therefore one read/write operation per clock cycle) and all their cells can be accessed and modified independently at once. The generated samples are simply sieved in such a manner that 24 are passed to the output stream, next $p - 24$ are rejected and so on. The p parameter value depends only on the chosen luxury level. The operation of sieving lowers generator performance simply by a factor of $24/p$. When the output stream buffer is full, data is sent to the Consumer process and new samples start to fill up the output buffer.

The results of the synthesis and implementation operations done using the Xilinx's ISE Design Suite 13.3 show that the generator core can run with a maximum frequency of 249 MHz, simultaneously occupying less than 5 % of the available logic resources (slices). Further information about the core itself in a more consistent form is shown in Table 3 at the end of this section.

3.2 VHDL

VHDL is one of the most frequently used languages for the description of digital systems such as FPGAs. VHDL is described in the IEEE Std. 1 076 standard. As for now, there were three revisions of this standard: the first one in 1993, the second one in 2003 and the last one in 2008. The most popular is the first revision from 1993 and it is denoted as the IEEE 1076–1993. It was used here to create the described RANLUX implementation. The data flow diagram of the VHDL implementation is shown in Figure 2, while a more defined block diagram is in Figure 3.

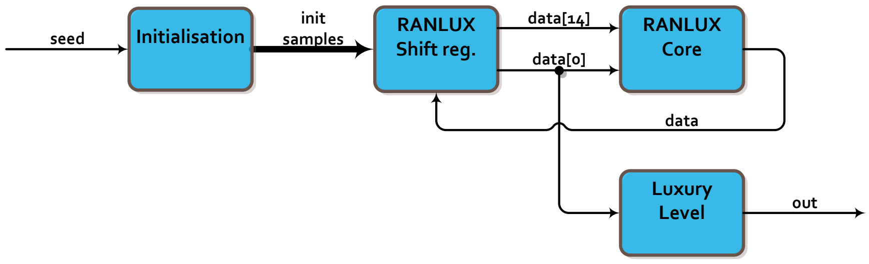

Figure 2. Data flow diagram of the RANLUX based PRNG implemented in VHDL

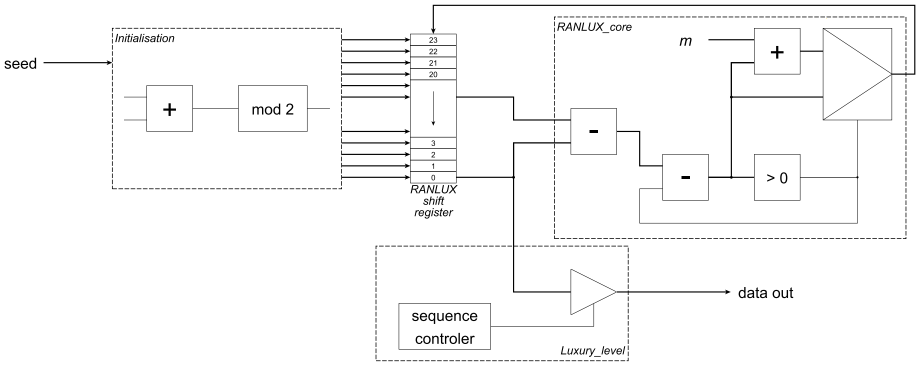

Figure 3. Block diagram of the RANLUX based PRNG implemented in VHDL

The block diagram is divided into four modules. The first of them is the Initialisation module. It has the same functionality as the RANLUX_Initialisation block in the Impulse C implementation case. The module provides initial values for the RANLUX algorithm. It means that it generates 576 bits of initial values using the dependency (3) using 31-bits seed value and it needs only one clock cycle for computation.

These achieved 576 bits are divided into 24 initial values that are transmitted to the RANLUX_Shift_register module at once. The shift register module is responsible for shifting data of one position per clock cycle and it outputs two values that are needed for the calculation of the next x_n number and the c_n carry bit.

For this calculation, Equation (2) is used and it is realised inside the RANLUX_Core module. The new value of x_n achieved from the core module is written to the shift register module in each clock cycle and the process of generation continues.

The last module is the Luxury_Level module. This module is responsible for luxury levels realisation. It takes all generated numbers from the RANLUX_Shift_register and decides which of them must be transmitted as the output pseudo-random numbers sequence. This decision depends on the used luxury level.

The results received from the tools of synthesis and implementation (ISE 13.1) for xc5vlx220-2 FPGA unit show that the RANLUX module can run with a maximum frequency of 414 MHz (Table 3). The RANLUX module consumed 3 % of the available resources of Xilinx xc5vlx220-2.

Parameter		Solution			
		Impulse C		VHDL	
f_{max}	[MHz]	249		414	
No. of registers	–	2 357	(1 %)	1 429	(1 %)
No. of LUTs	–	4 608	(3 %)	2 245	(1 %)
No. of BRAMs	–	1	(1 %)	0	(0 %)
No. of slices	–	1 846	(5 %)	1 216	(3 %)
Power Consumption	[W]	2.400		2.121	

Table 3. Detailed information about the designed RANLUX cores implemented in Xilinx's xc5vlx220-2 FPGA device

4 COMPARISON OF IMPLEMENTATIONS

All the detailed results are shown in Table 4. Both the GPP solutions were run on x86 architecture processors. The first one was Intel i5 2 520 M low power processor with a 2.5 GHz clock speed, while the second one was AMD high performance Phenom II X4 945 processor with a 3 GHz clock speed.

Exactly 100 measurements of time were taken to generate 100 MSa of usable PRNs at the given luxury level for the GPPs and both RPU implementations, and then averaged. The mean time to generate this 100 MSa at the given luxury level was then taken to count the throughput of those solutions.

As for the GPGPU results, they were borrowed from [8] and [22] exactly as they were. What should be noted is that the luxury levels in those tests were different than the ones for the GPPs and RPU.

The results presented in Table 4 and Figure 4 show that the FPGA based RANLUX generator is the most power efficient. It is 4 times more efficient than the GPP based solution and 290 times more efficient than the GPGPU based solution. The PRNs at luxury level 3 can be produced during 1 second, at the cost of 1 MHz of the processing unit clock speed. This 1 MHz is proportional to energy consumed by this unit.

Looking at the overall throughput of a single RANLUX instance, the GPP based, highly optimised solution in C language is the best. It has about 2.5 times higher throughput than the FPGA based one (at luxury level 3). The RPU used still has more than 67 times higher throughput than the GPGPU based solution.

The cause of the FPGA based RANLUX generator performance being lower than in case of the GPP based one is very simple. The RPU is working with a clock speed 12 times lower than the Intel i5 2 520 M and the performance obtained for luxury levels 1 – 4 was the highest possible at a 200 MHz clock speed. As for

		Platform							
		RPU[1]		GPP				GPGPU	
				C[2]		C++[3]			
		Impulse C	VHDL	Intel[6]	AMD[7]	Intel[6]	AMD[7]	OpenCL[4]	CUDA C[5]
Instance Throughput [MSa/s]	LL 0	134.4	–	550.7	444.9	120.2	118.0	1.23	0.32
	LL 1	100.6	100.0	265.6	239.2	64.8	64.4	0.85	0.19
	LL 2	49.8	49.5	121.3	112.3	33.2	33.4	0.53	0.10
	LL 3	21.6	21.6	55.2	47.8	15.0	15.2	0.27	–
	LL 4	12.4	12.3	31.5	27.2	8.7	8.8	0.17	–
Instance Efficiency [kSa/MHz]	LL 0	672	–	220.0	148.0	48.1	39.3	1.70	0.53
	LL 1	503	500.0	106.0	79.7	25.9	21.5	1.20	0.32
	LL 2	249	247.5	48.5	37.4	13.3	11.1	0.73	0.17
	LL 3	108	108.0	22.1	15.9	6.0	5.1	0.37	–
	LL 4	62	61.5	12.6	9.1	3.5	2.9	0.23	–

[1] Xilinx Virtex5 LX220-2, 200 MHz clock speed
[2] Lüscher's RANLUX v3.3, luxury levels as in Table 1
[3] Soyez's and Salam's RANLUX, luxury levels as in Table 1
[4] Demchik's RANLUX, luxury levels as in Table 2, ATI Radeon HD 5850
[5] Wende's RANLUX, luxury levels as in Table 2, nVidia Tesla C1060
[6] Intel i5 2 520 M, 2.5 GHz clock speed
[7] AMD Phenom II X4 945, 3 GHz clock speed

Table 4. Comparison of the different RANLUX implementations results

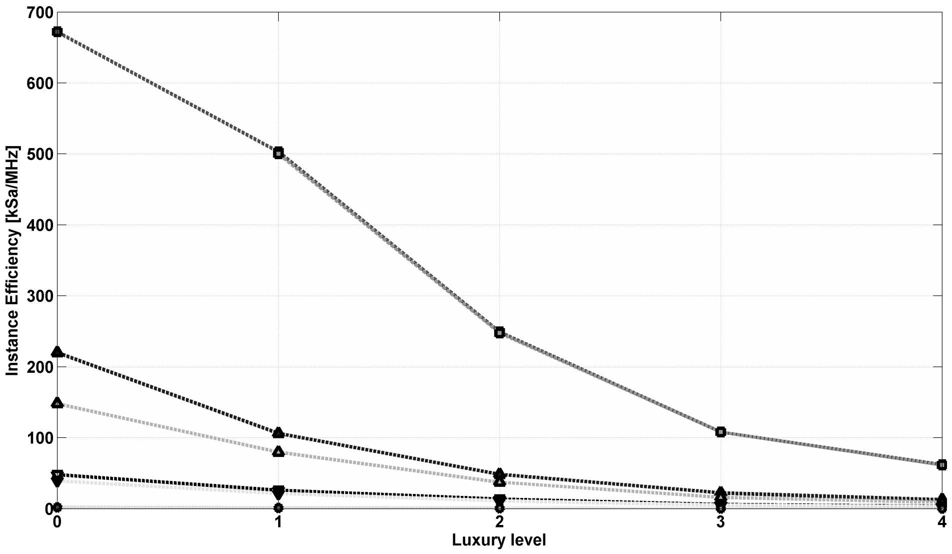

Figure 4. Efficiency of single instances of the RANLUX algorithm. Squares denote FPGA based implementation (dark – Impulse C, light – VHDL). Triangles denote GPP based implementations (upward-pointing triangle – C, downward-pointing triangle – C++, dark – Intel, light – AMD). Circles denote GPGPU based implementation (light – ATI, dark – nVidia).

luxury level 0, the limited HyperTransport v1.0 bus bandwidth was the cause of such a low result. Aldo DRC Corp. claims that the bandwidth should be at the level of 200 MTransfer/s, the real measurement was about 133 MTransfer/s.

5 CONCLUSIONS

The results show that the overall throughput of the multithread GPGPU based implementation of the RANLUX algorithm is the highest. The strength of the graphic card is in the high number of small computing cores. As shown in Table 4, the performance of a single core is poor, despite the fact that it works with a higher clock speed than presented RPU (e.g. nVidia Tesla C1060 CUDA Core clock is 602 MHz).

The GPP based performance depends on the quality of the algorithm implementation in the given programming language, as well as on the quality and age of the GPP used. The Intel i5 with a 20 % lower clock speed outperforms the AMD Phenom II for optimised C + ASM code and gives a similar performance in the case of the C++ implementation. The GPPs are characterised by a higher throughput per one RANLUX instance, with acceptable efficiency.

The FPGA based solutions are characterised by the best efficiency and the further possibility to implement more RANLUX cores inside the same FPGA unit. Such solutions would be similar to the multicore ones of GPPs and GPGPUs.

In the near future, the authors plan to introduce further improvements in the RANLUX algorithm hardware implementations, which should enable the increase in the performance of this pseudo-random number generator. At first, the authors want to use more than one RANLUX core in one FPGA. It was estimated that they can hold up to 25 RANLUX cores working in parallel with different seed values in one Xilinx xc5vlx220-2 device, without a heavy increase in power consumption.

The authors are currently working on the architecture, which allows computation of ten samples in one clock cycle, using a single RANLUX instance. The first tests prove that it increases the performance of the single RANLUX core implementation about 10 times compared to the solution presented in this article.

Acknowledgments

The authors would like to express their sincere appreciation to Dr. P. Russek from AGH–UST. He contributed significantly to the curriculum of the work. The authors would also like to thank E. Trexel and B. Dudwood from Impulse Accelerated Technology and M. Peterson form Synective Labs for their precious technical advice and guidance in comprehending language constructs and the right programming strategies of Impulse C.

This scientific work was made thanks to the "POWIEW" project. The project is co-funded by the European Regional Development Fund (ERDF) as a part of the Innovative Economy program.

APPENDIX A – Impulse C source code

Full version under GNU GPL available on (after August 2012):
`http://www.cyfronet.pl/projekty_badawcze/?a=powiew/centrum-komp`

RANLUX_Initialisation

```
void ranlux_init(co_int32 seed, co_int32 x[], co_uint1 *carry){
    #pragma CO INLINE

    int      i1;
    co_int32 iSeed = seed;
    co_int32 temp0;
    co_int32 temp1;
    co_int32 temp2;


    temp0 = co_bit_extract_u(seed, 0u, 24u);
    temp1 = co_bit_extract_u(seed, 24u, 7u);
    x[0]  = temp0;
    x[1]  = temp1;

    for(i1 = 1u;  i1 < MAGIC_1; i1++){
        #pragma CO UNROLL

        temp1 = co_bit_insert_u(temp1, 7u, 6u,
            co_bit_extract_u(temp0, 18u, 6u) ^
            co_bit_extract_u(temp0, 0u, 6u));

        temp1 = co_bit_insert_u(temp1, 13u, 11u,
            co_bit_extract_u(temp1, 0u, 11u) ^
            co_bit_extract_u(temp0, 6u, 11u));

        x[i1]    = temp1;

        temp2 = co_bit_extract_u(temp1, 11u, 7u) ^
            co_bit_extract_u(temp0, 17u, 7u);

        x[i1+1] = temp2;

        temp0   = temp1;
        temp1   = temp2;
    }

    temp1 = co_bit_insert_u(temp1, 7u, 6u,
        co_bit_extract_u(temp0, 18u, 6u) ^ co_bit_extract_u(temp0, 0u, 6u));

    x[MAGIC_1] = co_bit_insert_u(temp1, 13u, 11u,
        co_bit_extract_u(temp1, 0u, 11u) ^
        co_bit_extract_u(temp0, 6u, 11u));

    *carry = co_bit_extract_u(temp1, 31u, 1u);
}
```

RANLUX_Core

```
co_int32 ranlux_gen(co_int32 xLicznik, co_int32 xLicznik14, co_uint1 *carry)
{
    #pragma CO INLINE

    co_int32 delta;


    delta  = ISUB32(ISUB32(xLicznik14, xLicznik), *carry);

    *carry = co_bit_extract_u(delta, 31u, 1u);

    delta  = IADD32(delta, TF2TF);

    return co_bit_extract_u(delta, 0u, 24u);
}
```

RANLUX_Main

```
void ranlux(co_signal in, co_stream out){

    co_int32  seed;
    co_int32  nSample;
    co_int64  mSample;
    co_uint31 i;
    co_int32  x[MAGIC];
    co_int32  x1[MAGIC];
    co_uint1  carry = 0u;
    int       j;
    co_uint9  sieve = LEVEL;

    co_array_config(x, co_kind, "register");


    do {
        co_signal_wait(in, &seed);
        co_stream_open(out, O_WRONLY, INT_TYPE(STREAMWIDTH));

        co_par_break();

        ranlux_init_alt(seed, x1, &carry);


        for(j = 0; j < 24; j++){
            #pragma CO UNROLL

            nSample = x[j] = x1[j];
            mSample = (co_int64) nSample;
            co_stream_write(out, &mSample, sizeof(co_int64));
        }

        for(i = MAGIC; i < BORDER; i = UADD31(i, 1u)){
            #pragma CO PIPELINE

            nSample = ranlux_gen(x[0], x[14], &carry);
            for(j = 0; j < 23; j++){
                #pragma CO UNROLL

                x[j] = x[j+1];
            }
            x[MAGIC-1] = nSample;


            if(sieve < MAGIC){
                mSample = (co_int64) nSample;
                co_stream_write(out, &mSample, sizeof(co_int64));
            }

            sieve = UADD9(sieve, 1u);

            if(sieve >= LEVEL){
                sieve = 0u;
            }
        }

        co_stream_close(out);
    } while(1);
}
```

APPENDIX B – VHDL source code

RANLUX_Core

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;

entity basic_cell is
    port(
        Cin        : in  STD_LOGIC;
        x1         : in  STD_LOGIC_VECTOR(31 downto 0);
        x2         : in  STD_LOGIC_VECTOR(31 downto 0);
        RDY_x1     : in  STD_LOGIC;
        RDY_x2     : in  STD_LOGIC;
        RDY_Cin    : in  STD_LOGIC;
        Cout       : out STD_LOGIC;
        x_out      : out STD_LOGIC_VECTOR(31 downto 0);
        RDY_x_out  : out STD_LOGIC;
        RDY_Cout   : out STD_LOGIC
        );
end basic_cell;


architecture basic_cell of basic_cell is
begin

    process(x1, x2, Cin, RDY_x1, RDY_x2, RDY_Cin)
        variable Cin_var  : STD_LOGIC;
        variable delta    : STD_LOGIC_VECTOR(31 downto 0);
    begin
        if RDY_Cin = '0' then
            Cin_var := '0';
        else
            Cin_var := Cin;
        end if;

        if RDY_x1 = '1' and RDY_x2 = '1' then
            delta := x1 - x2 - Cin_var;

            if delta >= 0 then
                Cout      <= '0';

                x_out     <= delta;

                RDY_Cout  <= '1';
                RDY_x_out <= '1';
            else
                Cout      <= '1';
                x_out     <= delta + "00000001000000000000000000000000";

                RDY_Cout  <='1';
                RDY_x_out <='1';
            end if;
        else
            RDY_x_out <= '0';
            RDY_Cout  <= '0';
        end if;
    end process;
end basic_cell;
```

REFERENCES

[1] Lüscher, M.: A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations. Computer Physics Communications, Vol. 79, 1994, No. 1, pp. 100–110.

[2] Marsaglia, G.—Zaman, A.: A New Class of Random Number Generators. Annals of Applied Probability, Vol. 1, 1991, pp. 462–480.

[3] James, F.: RANLUX: A Fortran Implementation of the High-Quality Pseudo-Random Number Generator of Lüscher. Computer Physics Communications, Vol. 79, 1994, No. 1, pp. 111–114.

[4] James, F.: Finally, a Theory of Random Number Generation. In: D. Fotiadis and C. Massalas (Eds.): Scattering and Biomedical Engineering: Modeling and Applications, Proceedings of the Fifth International Workshop on Mathematical Methods in Scatter, Corfu, October 2001, pp. 114–121.

[5] Hamilton, K. G.—James, F.: Acceleration of RANLUX. Computer Physics Communications, Vol. 101, 1997, No. 3, pp. 241–248.

[6] Hamilton, K. G.: Assembler RANLUX for PCs. Computer Physics Communications, Vol. 101, 1997, No. 3, pp. 249–253.

[7] Lüscher, M.: Algorithms used in RANLUX v3.0. Available on: `http://luscher.web.cern.ch/luscher/ranlux/notes.pdf`.

[8] Demchik, V.: Pseudo-Random Number Generators for Monte Carlo Simulations on ATI Graphics Processing Units. Computer Physics Communications, Vol. 182, 2011, No. 3, pp. 692–705.

[9] Nikoleisen, I.: Quantum Monte Carlo Analysis of Bose–Einstein Condensation. AMD Fusion^{11} Developer Summit 2011.

[10] Impulse Accelerated Technologies Web site. Available on: `http://www.impulseaccelerated.com`.

[11] DRC Computer Corp. Accelium^{TM} Appliance Platform Product DataSheet, PO A 7–08, 2012. `http://drccomputer.com/pdfs/DRC_Accelium_Appliance_Platform.pdf`.

[12] DRC Computer Corp. Accelium^{TM} Coprocessors Product DataSheet, DS AC 7–08, 2012. `http://drccomputer.com/pdfs/DRC_Accelium_Coprocessors.pdf`

[13] Xilinx Web site. Available on: `http://www.xilinx.com`.

[14] HyperTransport Consortium Web site. Available on: `http://www.hypertransport.org`.

[15] Bowen, M.: Handel-C: Language Reference Manual. 2012, Available on: `http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel\%20C.PDF`.

[16] Pietroń, M.—Russek, P.—Wiatr, K.: Loop Profiling Tool for HPC Code Inspection as an Efficient Method of FPGA Based Acceleration. International Journal of Applied Mathematics and Computer Science, Vol. 20, 2010, No. 3, pp. 581–589.

[17] ACC CYFRONET AGH Web site. Available on: `http://cyfro.net/en/`.

[18] Giorgi, M. et al.: SuperB: A High-Luminosity Heavy Flavour Factory. INFN, Conceptual Design Report, March 2007. Available on: `http://www.pi.infn.it/SuperB/?q=CDR`.

[19] Chwastowski, J.—Chrząszcz, M.—Lesiak, T.: The SuperB Project: A Window to the Matter-Antimatter Asymmetry and a Challenging Computing Platform. In: Proceedings of the 5th ACC CYFRONET AGH Users Conference, Zakopane, March 2012, p. 65.

[20] RANLUX homepage. Available on: `http://luscher.web.cern.ch/luscher/ranlux/`.

[21] SISCone Jet Algorithm homepage. Available on: `http://siscone.hepforge.org/`.

[22] Wende, F.: Simulation of Spin Models on Nvidia Graphics Cards Using CUDA. Available on: `http://edoc.hu-berlin.de/master/wende-florian-2010-10-20/PDF/wende.pdf`.

[23] nVidia Web site. Available on: `www.nvidia.com`.

[24] Curd, D.: Power Consumption in 65 nm FPGAs. Xilinx White Paper: Virtex-5 FPGAs, WP246 (v1.2), 2007. Available on: `http://www.xilinx.com/support/documentation/white_papers/wp246.pdf`.

[25] Gupta, S. et al.: CAD Techniques for Power Optimization in Virtex-5 FPGAs. In: IEEE Custom Integrated Circuits Conference, San Jose, September 2007, pp. 85–88.

Agnieszka Dąbrowska-Boruch received M. Sc. and Ph. D. degrees in electronics from the AGH University of Science and Technology (AGH–UST), Kraków, Poland, in 2002 and 2007, respectively. Since 2004 she is with the Department of Electronics, AGH-UST, Kraków, Poland. She has published over 30 papers in journals and conferences and also one book: "FPGA implementation of real-time video coding in MPEG-2 standard" (Warszawa: Akademicka Oficyna Wydawnicza EXIT, 2008). Her research interests include image compression, real time systems, reconfigurable systems and devices, hardware acceleration of computations.

Grzegorz Gancarczyk received the M. Sc. degree in electronics from the AGH-UST, Kraków, Poland, in 2009. Since 2009 he is with the Academic Computer Centre (ACC) CYFRONET AGH, Kraków, Poland and now also with the Department of Electronics, AGH-UST, Kraków, Poland. His research interests include stochastic processes, statistics, phenomenon of noise, digital signal processing and hardware acceleration of numerical methods.

Kazimierz Wiatr received M. Sc. and Ph. D. degrees in electrical engineering from the AGH–UST, Kraków, Poland, in 1980 and 1987, respectively, D. Hab. (habilitation) degree in electronics from the University of Technology of Łódz, Łódz, Poland, in 1999 and the Professor degree in 2002. Since 1980 he works at the Department of Electronics, AGH-UST, Kraków, Poland. He is the Head of Reconfigurable Computing Systems Group. Since 2004 he is the Director of the ACC CYFRONET AGH. Since 2006 he serves as chairman of the board of PIONIER – Polish Optical Internet – Consortium. Between 1998 and 2002 he was the adviser to the Prime Minister of Poland on "education and upbringing of the young generation". He managed 9 Polish Scientific Research Committee research grants. His works resulted in over 200 publications, 19 books, 5 patents and 35 industrial implementations. He achieved Polish Science and Higher Education Minister's Award. He has been involved with youth education for more than 30 years. He is one of the founders of the Polish independent scouting movement. His research interests include educational issues, processes automation, image systems, multiprocessor and many core systems, reconfigurable devices and hardware methods of calculations accelerating.