ISSN 0361-7688, Programming and Computer Software, 2017, Vol. 43, No. 3, pp. 145–160. © Pleiades Publishing, Ltd., 2017.
Original Russian Text © L.Yu. Barash, M.S. Gus'kova, L.N. Shchur, 2017, published in Programmirovanie, 2017, Vol. 43, No. 3.

Employing AVX Vectorization to Improve the Performance of Random Number Generators

L. Yu. Barash[a,b,*], M. S. Gus'kova[b,c,], and L. N. Shchur[a,b,c,***]**

[a] *Landau Institute for Theoretical Physics, Russian Academy of Sciences, ul. Kosygina 2, Moscow, 119334 Russia*

[b] *Science Center in Chernogolovka, Russian Academy of Sciences, Institutskii pr. 8, Chernogolovka, 142432 Russia*

[c] *Moscow Institute of Electronics and Mathematics, National Research University Higher School of Economics, ul. Malaya Pionerskaya 12, Moscow, 115054 Russia*

**e-mail: barash@chg.ru*

***e-mail: maria.guskova@rambler.ru*

****e-mail: lev@landau.ac.ru*

Received November 13, 2016

Abstract—By the example of the RNGAVXLIB random number generator library, this paper considers some approaches to employing AVX vectorization for calculation speedup. The RNGAVXLIB library contains AVX implementations of modern generators and the routines allowing one to initialize up to 10^{19} independent random number streams. The AVX implementations yield exactly the same pseudorandom sequences as the original algorithms do, while being up to 40 times faster than the ANSI C implementations.

DOI: 10.1134/S0361768817030033

1. INTRODUCTION: PSEUDORANDOM NUMBER GENERATORS

With the development and widespread use of powerful computers, Monte-Carlo methods and molecular dynamics methods became intensively employed for solving a great many problems in various fields of physics, including quantum physics [1], statistical physics [2, 3], nuclear physics [4], quantum chemistry [5], and materials sciences [6]. Implementing these methods requires generating uniformly distributed random numbers. For this purpose, pseudorandom number generators (PRNGs) are employed. A sequence yielded by an efficient and reliable PRNG possesses properties of a random sequence, though being generated in a deterministic way, i.e., being pseudorandom. Improving the performance of PRNGs is rather important because, in many cases (e.g., involving the use of Monte-Carlo methods), random number generation takes a considerable amount of the computation time. The most popular PRNGs can be divided into two main groups: linear congruential generators (LCGs) and linear-feedback shift register (LFSR) generators.

1.1. Linear Congruential Generators

The LCG was originally proposed by Lehmer [7]. Upon setting a value x_0, a pseudorandom sequence $(x_0, x_1, \ldots, x_n, \ldots)$ is calculated by the formula $x_{n+1} = (ax_n + c) \bmod M$, where a is the multiplier, c is the increment, and M is the modulus. The maximum period of this sequence cannot exceed M. The LCG has two significant drawbacks. The first one is the algorithmic constraint on the maximum period, which cannot exceed a machine-size integer (in this case, the sequence of length $2^{32} \approx 4 \times 10^9$ runs out on modern computers in a few seconds). The second drawback is that the LCG is useless for working with random vectors in the n-dimensional space when $n > 1$ because of a bad lattice structure of the vectors generated, which are all positioned on a set of parallel hyperplanes [8, 9].

1.2. Linear-Feedback Shift Register (LFSR) Generators

The LFSR generator uses the properties of linear modulo 2 operations on bits of x_n. Suppose we have a sequence $x_n = (a_1 x_{n-1} + \ldots + a_k x_{n-k}) \bmod 2$, which is a linear recurrence relation in the field Z_2 consisting of two elements (zero and one). It is called a shift register and has a period $p = 2^k - 1$ if and only if the polynomial $P(z) = z^k - a_1 z^{k-1} - \ldots - a_k$, also referred to as a characteristic polynomial of the recurrence relation, is a primitive one [10]. The LFSR is a generator with the

output sequence $u_n = \sum_{i=1}^{L} x_{ns+i-1}2^{-i}$, where s (step size) and L (word length) are positive integers. This definition suggests that LFSR generators are fast. They have a very long period (provided that primitive polynomials are selected properly). However, these generators employ certain correlations that can cause systematic errors in Monte–Carlo calculations. These properties were thoroughly analyzed in [11–18, 44].

Classical LCGs and LFSRs fail many empirical tests (see Section 4.5.4 in [19], as well as [20]) and, therefore, are useful only in particular cases rather than in a wide class of scientific and engineering problems. Modern modifications and generalizations of classical LCGs and LFSRs have far better statistical properties, for example, Mersenne Twister generator [21], combined multiple recursive random number generators [22], combined LFSR generators [23, 24], and generators based on the parallel evolution of toral automorphisms [25–29]. Here, we should also mention recently proposed counter-based generators [33, 34].

There are many problems solved by means of Monte–Carlo simulation, where random number generation takes a considerable amount of computational time. Therefore, it is important to speed up the random number generation process by employing modern algorithm optimization technologies. For this purpose, we use the Advanced Vector Extensions 2 (AVX2) technology, which is supported by presently-available Intel (since 2013, the Intel Haswell microarchitecture) and AMD (since 2014, the Streamroller Family 15h microarchitecture) processors. It is shown that vectorization makes it possible to significantly improve the performance of PRNGs.

The methods described in this paper were used to develop the RNGAVXLIB library [35], which contains implementations of the following modern and reliable PRNGs: MT19937 [21], MRG32K3A [22], LFSR113 [24], GM19, GM31, GM61 [27], GM29, GM55, GQ58.1, GQ58.3, and GQ58.4 [25, 29]. The library contains implementations written in ANSI C, those using SSE vectorization, and those using the AVX2 extension.

The concept of a PRNG can be formalized as follows: a generator is a structure $\mathcal{G} = (S, s_0, T, U, G)$, where S is a finite *set of states*, $s_0 \in S$ is an initial state, $T : S \to S$ is a *transition function*, U is a finite set of output symbols, and $G : S \to U$ is an *output function* [32]. Initially, the generator is in the state s_0; the generator changes its state at each step n by calculating the values $s_n = T(s_{n-1})$ and $u_n = G(s_n)$. The values u_n at the generator output are called *observable values* or, simply, output *pseudorandom numbers*. Below, we briefly describe the generator algorithms implemented in the RNGAVXLIB library.

1.3. MT19937 Algorithm

The MT19937 algorithm is the Mersenne Twister (MT) generator, which was developed by Matsumoto and Tishimura [21] as a modification and generalization of the LFSR method. The MT generates 32-bit vectors by using the recurrence relation

$$x_{k+n} := x_{k+m} \oplus (x_k^u | x_{k+1}^l)A, \tag{1}$$

where $k = 0, 1, \ldots$. The MT19937 has the parameters $n = 624$ and $m = 397$. Here $(x_k^u | x_{k+1}^l)$ is a 32-bit integer obtained by joining the most significant bit x_k and 31 least significant bits x_{k+1}, $\oplus$ is the XOR operation, and A is the companion matrix such that

$$xA = \begin{cases} x \gg 1, & \text{if the least significant} \\ & \text{bit of the vector } x \text{ is } 0, \\ (x \gg 1) \otimes a, & \text{if the least significant} \\ & \text{bit of the vector } x \text{ is } 1, \end{cases}$$

where the vector a is a 32-bit integer constant (2567483615). Here, $x \gg i$ represents the right shift of x by i bits, while $x \ll i$ represents the left shift of x by i bits.

To improve the equidistribution of the output sequences, each word is multiplied by the invertible matrix T: $x \mapsto z = xT$ through the successive execution of the following transformations [21]:

$$y := x \oplus (x \gg u), \tag{2}$$

$$y := y \oplus ((y \ll s)\, AND\, b), \tag{3}$$

$$y := y \oplus ((y \ll t)\, AND\, c), \tag{4}$$

$$z := y \oplus (y \gg l), \tag{5}$$

where $u = 11$, $s = 7$, $t = 15$, $l = 18$, $b = 2636928640$, and $c = 4022730752$.

The MT19937 is a generator with a very long period $(2^{19937} - 1)$. For this generator, the 623-dimensional equidistribution property holds; in practice, this means that the MT19937 can uniformly fill out spaces with up to 623 dimensions, which makes it attractive for use in multidimensional integration. Like most generators, the MT is sensitive to bad initialization, which makes correct initialization very important.

1.4. MRG32K3A Algorithm

The MRG32K3A algorithm is a combined MRG generator, which was found in [22] by exhaustive search of MRG parameters. The MRG32K3A is a combination of two linear generators.

Its transition function is given by the recurrence relations

$$x_{i+3} = (ax_i + bx_{i+1}) \bmod m_1, \tag{6}$$

$$y_{x+3} = (cy_i + dy_{i+2}) \bmod m_2, \tag{7}$$

Table 1. Parameters of the generators from the RNGAVXLIB library [35]

Generator	k	q	g	v	Period length	Dimension of approx. equidistribution
GM19	15	28	$2^{19}-1$	1	2.7×10^{11}	129
GM29	4	2	$2^{29}-3$	1	2.8×10^{17}	200
GM31	11	14	$2^{31}-1$	1	4.6×10^{18}	210
GM55	256	176	$16(2^{51}-129)$	4	$\geq5.1\times10^{30}$	350
GM61	24	74	$2^{61}-1$	1	5.3×10^{36}	415
GQ58.1	8	48	$2^{29}(2^{29}-3)$	1	$\geq2.8\times10^{17}$	200
GQ58.3	8	48	$2^{29}(2^{29}-3)$	3	$\geq2.8\times10^{17}$	200
GQ58.4	8	48	$2^{29}(2^{29}-3)$	4	$\geq2.8\times10^{17}$	200
LFSR113	–	–	–	–	1.0×10^{34}	30
MRG32K3A	–	–	–	–	3.1×10^{57}	45
MT19937	–	–	–	–	4.3×10^{6001}	623

$$u_{i+3} = (x_{i+3} - y_{i+3}) \bmod m_1. \quad (8)$$

Here, $a = -810728$. $b = 1403580$. $c = -1370589$, $d = 527612$, $m_1 = 2^{32} - 209$, and $m_2 = 2^{32} - 22853$.

The initial state of the first generator is described by the variables x_0, x_1, and x_2, while that of the second generator is described by the variables y_0, y_1, and y_2. At each step, the output value u_n is calculated. The period of the MRG32K3A is approximately 3.1×10^{57}.

1.5. LFSR113 Algorithm

The LFSR113 generator is a combination of the following LFSRs [23, 24]:

1. $x_{1,n} = x_{1,n-31} \oplus x_{1,n-25}$, $s_1 = 18$,
2. $x_{2,n} = x_{2,n-29} \oplus x_{2,n-27}$, $s_2 = 2$,
3. $x_{3,n} = x_{3,n-28} \oplus x_{3,n-15}$, $s_3 = 7$,
4. $x_{4,n} = x_{4,n-25} \oplus x_{4,n-22}$, $s_4 = 13$.

For these LFSRs, the output sequences $u_{i,n}$ are given by the relation $u_{i,n} = \sum_{j=1}^{L} x_{i,ns_i+j-1}2^{-j}$, where s_i is the step size and L is the word length. In turn, the output sequence of the combined generator is defined as $u_n = u_{1,n} \oplus u_{2,n} \oplus u_{3,n} \oplus u_{4,n}$. The word length of the LFSR113 is $L2$, while its period is approximately 2^{113}.

1.6. Generators Based on Parallel Evolution of Toral Automorphisms: GM19, GM31, GM61, GM29, GM55, GQ58.1, GQ58.3, and GQ58.4

In [25–27], an algorithm was proposed for constructing PRNGs based on the parallel evolution of a torus transformations ensemble. The set of states for such a PRNG is defined as $S = L^S$, where $L = \{0,1,\ldots,g-1\}^2$, while g and s are integers ($s \leq 32$ and g is a product of a sufficiently large prime integer by a certain power of two). Thus, a generator state contains s pairs of integers from the set $\{0,1,\ldots,g-1\}$, which are denoted (at the nth step of the generation process) as $x_i^{(n-1)}, x_i^{(n-2)} \in \{0,1,\ldots,g-1\}$, $i = 0,1,\ldots,s-1$. The transition function of this generator is given by the recurrence relation

$$x_i^{(n)} = kx_i^{(n-1)} - qx_i^{n-2} (\bmod g), \quad i = 0,1,\ldots,s-1. \quad (9)$$

Here, k and q are integers. It can be shown that this recurrence relation describes the dynamics of the x-coordinates of s points for the automorphism of a two-dimensional torus. The output pseudorandom number appears as

$$a^{(n)} = \sum_{i=0}^{s-1} \left\lfloor 2^v x_i^{(n)}/g \right\rfloor \times 2^{iv}, \quad (10)$$

i.e., it takes v bits from each recurrence relation.

Table 1 shows the parameters of the generators based on the automorphism of a two-dimensional torus, which were proposed in [25–27]. The parameters were selected so as to guarantee the equidistribution property in dimensions up to a logarithm of g, as well as providing as large generator period as possible [25].

2. CAPABILITIES AND FUNCTION CALL INTERFACE OF THE RNGAVXLIB LIBRARY

The source code of the RNGAVXLIB library can be found in the program library of the *Computer Physics Communications* journal [35]. To run the AVX implementations, the Intel or AMD processor supporting the AVX2 instruction set is required. The SSE implementations require the Intel or AMD processor supporting the SSE2 instruction set. To run the SSE version of the LFSR113 generator, the processor must support SSE4.1. The library can automatically select

Table 2. Function interface of the RNGAVXLIB library

Function	Description
void rng_init_(rng_state* state);	Initializing the generator;
void rng_init_sequence_(rng_state* state, unsigned long long SequenceNumber);	Initializing an independent random number stream with the number SequenceNumber;
void rng_skipahead_(rng_state* state, unsigned long long offset);	Fast skipping of output values;
unsigned int rng_generate_(rng_state* state);	Generating 32-bit random integers (the processor instruction set is selected automatically at the compilation stage);
float rng_generate_uniform_float_(rng_state* state);	Generating random reals uniformly distributed on the interval [0,1) (the processor instruction set is selected automatically at the compilation stage);
unsigned int rng_ansi_generate_(rng_state* state);	Generating 32-bit random integers by using ANSI C only;
float rng_ansi_generate_uniform__oat_(rng_state* state);	Generating random reals uniformly distributed on the interval [0,1) by using ANSI C only;
unsigned int rng_sse_generate_(rng_state* state);	Generating 32-bit random integers by using SSE vectorization;
float rng_sse_generate_uniform_float_(rng_state*state);	Generating random reals uniformly distributed on the interval [0,1) by using SSE vectorization;
unsigned int rng_avx$_g$*enerate*_(*rng_state** state);	Generating 32-bit random integers by using AVX vectorization;
float rng_avx_generate_uniform_float_(rng_state* state);	Generating random reals uniformly distributed on the interval [0,1) by using AVX vectorization;
void rng_print_state_(rng_state* state);	Displaying the generator state.

the instruction set supported by a given processor. In the case of a heterogeneous computer cluster, the user has to manually select the instruction set supported by all processors in the cluster. The RNGAVXLIB library also makes it possible to jump ahead in a pseudorandom sequence, as well as to initialize independent random number streams with the block splitting method. The library supports C and FORTRAN.

For all the generators, the RNGAVXLIB library provides the functions listed in Table 2 (note that **rng** stands for the name of a particular generator).

The functions rng_generate_ and rng_generate_uniform_float_ use SSE or AVX only if it is supported by the processor. The corresponding instruction set is selected automatically at the compilation stage by using the compiler option -march=native.

For some generators, function calls have certain specifics. For example, the function

void mt19937_skipahead_(mt19937_state* state, unsigned long long a, unsigned b);

skips $N = a \times 2^b$ numbers (where $N < 2^{512}$), while the function

void gm55_skipahead_(gm55_state* state, unsigned long long offset64, unsigned long long offset0);

skips $N = 2^{64} \times \text{offset}\,64 + \text{offset}\,0$ numbers. The function call interface with a detailed parameter lists can be found in the header files (located in the directory **include**).

The GNU Fortran has no compiler directives for data alignment, which is required for vectorization. The Intel Fortran, however, provides such directives, e.g., **!dir$ attributes align:32**. By default, the GNU Fortran aligns all variables along 16-byte boundaries, which is sufficient for SSE but not sufficient for AVX. We found that the additional instruction SAVE state in the GNU Fortran enables, among other things, data alignment along 32-byte boundaries, which makes it possible to use the AVX implementations from Fortran. We have tested this feature on different processors and different versions of Linux.

For the LFSR113, the RNGAVXLIB library enables the simultaneous generation of two independent random sequences by using AVX vectorization:

void lfsr113_avx_generate_two_(lfsr113_state* state, unsigned * out1, unsigned *out2);

This is the fastest way we know to generate random numbers with the LFSR113 on the processors supporting the AVX2 technology (see also Section 5). The procedure **lfsr113_skipahead2_** can be used to jump ahead in the second sequence.

Some of the generators have several versions of the procedure **rng_init_sequence_**, for example, **rng_init_short_sequence_**, **rng_init_medium_sequence_**, and **rng_init_long_sequence_** (see [30, 31]). The maximum number of streams and the maximum length of a sequence for parallel random number generation are specified in [30, 31]. The algorithms used to jump ahead in the RNG sequence and the algo-

Table 3. Implementations of the random number generation algorithm for the GM55

AVX2 implementation

```
#define g      36028797018961904ULL
typedef unsigned long long It;
typedef struct{ It
  xN [8] _attribute_  ((aligned(32))),
  xP [8] _attribute_  ((aligned(32)));
}  gm55_state;
It avx_Consts [16] _attribute_
((aligned(32))) = {11*g, 11*g, 11*g,
11*g, 2064ULL, 2064ULL, 2064ULL,
2064ULL, 36028792732385279ULL,
36028792732385279ULL,
36028792732385279ULL,
36028792732385279ULL, g, g, g, g};
unsigned int gm55_avx_generate_(gm55_state*state){.
unsigned output;
asm volatile(
/* Evaluating the transition function: */
  "vmovaps (%3),%%ymm0\n" \
  "vmovaps (%2),%%ymm1\n" \
  "vmovaps (%1), %%ymm4\n" \
  "vmovaps %%ymm4, (%2)\n" \
  "vpsllq $4,%%ymm4,%%ymm4\n" \
  "vpaddq %%ymm0,%%ymm4,%%ymm4\n" \
  "vmovaps %%ymml,%%ymm2\n" \
  "vpaddq %%ymml,%%ymm2,%%ymm2\n" \
  "vpaddq %%ymml,%%ymm2,%%ymm2\n" \
  "vpsllq $3,%%ymml,%%ymm1\n" \
  "vpaddq %%ymml,%%ymm2,%%ymm2\n" \
  "vpsubq %%ymm2,%%ymm4,%%ymm4\n" \
  "vmovaps %%ymm4,%%ymm2\n" \
/* Calculating the remainders on dividing by g: */
  "vpsrlq $51,%%ymm2,%%ymm2\n" \
  "vmovaps %%ymm2,%%ymm3\n" \
  "vpsllq $7,%%ymm3,%%ymm3\n" \
  "vpaddq %%ymm2,%%ymm3,%%ymm3\n" \
  "vpsllq $51,%%ymm2,%%ymm2\n" \
  "vpsubq %%ymm2,%%ymm4,%%ymm4\n" \
  "vpaddq %%ymm3,%%ymm4,%%ymm4\n" \
  "vpsllq $4,%%ymm4,%%ymm4\n" \
  "vmovaps %%ymm4,%%ymml\n" \
  "vpaddq 32(%3), %%ymm1,%%ymm1\n" \
  "vpshufd $245,%%ymm1,%%ymm3\n" \
  "vpcmpgtd 64(%3),%%ymm3,%%ymm3\n" \
  "vpand   96(%3) ,%%ymm3,%%ymm3\n" \
  "vpsubq  %%ymm3,%%ymm4,%%ymm4\n" \
  "vmovaps %%ymm4, (%1) \n" \
             [...]
/* Calculating the output value: */
  "vpsrlq $51,%%ymm4,%%ymm4\n" \
  "vpsrlq $51,%%ymm6,%%ymm6\n" \
  "vpackssdw %%ymm6,%%ymm4,%%ymm4\n" \
  "vpermpd $216, %%ynm4, %%ynm4\n"\
  "vpackssdw %%ymm4,%%ymm4,%%ymm4\n" \
```

ANSI C implementation

```
#define k 256
#define q 176
#define g      36028797018961904ULL
#define gdivl6 2251799813685119ULL
typedef unsigned long long It;
typedef struct{ It xN [8], xP [8];
}  gm55_state;
It gm55_mod_g(lt x){//returns x (mod g)
  It F,G;
  F = (x>>55);
  G = x-(F<<55)+(2064*F);
  return ((G>=g) ? (G-g) : G);
}
unsigned int
gm55_ansi_generate_(gm55_state*
state){
  unsigned int sum=0; int i; It temp;
  for(i=0;i<8;i++){
    temp=gm55_mod_g((
(state->xN[i])<<8)+q*(g-state-
>xP[i]));
    state->xP[i]=state->xN[i];
    state->xN[i]=temp;
      sum+=
((temp/gdivl6)<<((i<4)?(8*i) : (8*i-28)));
  }
  return sum;
}
```

Table 3. (Contd.)

AVX2 implementation	ANSI C implementation

`"vpermpd $216, %%ynm4, %%ynm4\n"\`
`"vpacksswb %%ymm4,%%ymm4,%%ymm4\n" \`
`"vmovaps %%ymm4,%%ymm0\n." \`
`"vpsrldq $4,%%ymm0,%%ymm0\n" \`
`"vpslld $4,%%ymm0,%%ymm0\n" \`
`"vpxor %%ymm0,%%ymm4,%%ymm4\n" \`
`"vmovd %%xmm4,%0\n" \`
`"":"=&r"(output):"r"(state->xN),`
`"r"(state->xP),"r"(avx_Consts));`
`return output;`
`}`	

rithms for initializing parallel pseudorandom number streams are described in [31, 37].

3. TESTING PRNGs FROM DIFFERENT SOFTWARE PACKAGES

In [31, 42], the comparative analysis was carried out of presently-available libraries for generating random numbers and parallel random number streams, including GNU Scientific Library [38], Intel MKL Library [39], RNGSSELIB [30], SPRNG [40], and TRNG [41]. Moreover, in [31], the performance capabilities of the PRNGs provided by these libraries were compared (see Table 1 in [31]). The performance was estimated based on the following criteria: generation speed, period length, equidistribution dimension, and number of crushes (number of failed tests from the TestU01 package [20]: SmallCrush, Crush, and BigCrush). The crush criterion characterizes statistical properties of the generators. The results obtained in [31] correlate well with the results of statistical testing presented in [20]. These results show that the generators implemented in the RNGAVXLIB library [35] possess the best properties. The results also demonstrate that using SSE instructions and 128-bit XMM registers provides a considerable speed-up of calculations and, therefore, increase in the efficiency of random number generation. Further speed-up can be achieved with AVX vectorization (see Section 8). Generation of parallel random number streams was discussed in [30, 31, 37, 42, 43].

4. AVX ALGORITHMS FOR THE GM AND GQ GENERATORS

In terms of basic characteristics, the GM55 stands out among the other generators of the GM and GQ families (GM19, GM31, GM61, GM29, GM55, GQ58.1, GQ58.3, AND GQ58.4) as possessing a long period and high speed of pseudorandom number generation (see Sections 1 and 8). Therefore, we discuss the specifics of AVX implementation of the GM and GQ generators by the example of the GM55. Table 3 shows the implementations of the AVX (left column) and ANSI C (right column) algorithms for the GM55 generator. The AVX algorithm does the same thing as the ANSI C algorithm, but all its operations are applied to a vector of four numbers. The detailed description of the AVX2 instructions can be found, for example, in [36].

Let us consider the AVX algorithm in detail. First, the algorithm calculates the values $z_i = 16\times x_i^{n-1} + 11\times g - 11\times x_i^{n-2}$ where $i = 0,\ldots,3$. The value $11\times g$ is added to make the residual positive. For calculations, the algorithm uses the equalities $16\times x_i^{n-1} = (x_i^{n-1} \ll 4)$ and $11\times x_i^{n-2} = x_i^{n-2} + x_i^{n-2} + x_i^{n-2} + (x_i^{n-2} \ll 3)$. Upon multiplying by 16, we obtain the values $256\times x_i^{n-1} - 176\times(g - x_i^{n-2})$, which are positive and have the same remainders on dividing by g as $256\times x_i^{n-1} - 176\times x_i^{n-2}$. Thereby, we evaluate recurrence relation (9), where $k = 256$; $q = 176$; $g = 16\times(2^{51} - 129)$. Next, the remainders on dividing $16\times z_i$ by g are found. With the right logical shift by 51 bits, the values $[z_i/2^{51}]$ $(i = 0,\ldots,3)$ are placed into the registers YMM2 and YMM3. Taking into account that the left logical shift by 7 bits is equivalent to multiplying by 128, the algorithm calculates the values $z_i - 2^{51}\times[z_i/2^{51}] + 129\times[z_i/2^{51}]$. After the left shift by 4 bits, the register YMM4 contains the values $y_i = 16\times z_i - g\times[z_i/2^{51}]$, $i = 0,\ldots 3$. These values are nonnegative and have the same remainders on dividing by g as $16\times z_i$. The values z_i cannot exceed $(16+11)\times g$, so $z_i/2^{51} < 432$. Hence, the values y_i cannot exceed $2^{55} + 891648$) because

$$y_i = 16\times z_i - 2^{55}\times[16\times z_i/2^{55}] + 16\times 129[z_i/2^{51}]$$

$$= (16\times z_i)(\mathrm{mod}\, 2^{55}) + 2064\times[z_i/2^{51}] < 2^{55} + 891648.$$

Next, if $y_i \geq g$, then the values of all 64-bit YMM4 banks are reduced by g. Since there is no instruction for simultaneous comparison among 64-bit integers from the YMM registers, we use the instruction VPCMPGTD that simultaneously compares 32-bit integers. For this purpose, using the instruction VPSHUFD, we place into the register YMM3 two identical 32-bit values $(y_i + 2064) \gg 32$ for all $i = 0,\ldots,3$. Then, these 32-bit values are compared with $(2^{23} - 1)$ by using the instruction VPCMPGTD.

The same manipulations are performed for $i = 4,\ldots,7$, and the result is placed into the register YMM6. This part of the program (see the left column in Table 6) is replaced by ellipsis.

The remaining part of the program evaluates output function (10) of the generator as follows. The instructions **vpsrlq** retain only four most significant bits of each 64-bit number; these 4-bit numbers are denoted by $t_0, t_1, \ldots, t_7$. The instruction **tbfvpackssdw** packs double words into words so that the register YMM4 contains the numbers $t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7$. The instruction **vpermpd** swaps the positions of the second and third 64-bit blocks in the register YMM4 $(t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7)$. Once applied to the register YMM4 a second time, the instruction **vpackssdw** leaves therein 16 numbers $t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7$. The instruction **vpermpd** again swaps the positions of the second and third 64-bit blocks, and the operation **extbfvpacksswb** packs 16-bit words into bytes. As a result, each 64-bit block of the register YMM4 contains eight numbers $t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7$. These numbers are distributed over 32 least significant bits as follows: YMM4[0 : 3] = t_0, YMM4[4 : 7] = 0, YMM4[8 : 11] = t_1, YMM4[12 : 15] = 0, YMM4[16 : 19] = t_2, YMM4[20 : 23] = 0, YMM4[24 : 27] = t_3, and YMM4[28 : 31] = 0. Next, the instructions **vpsrldq** and **vpssld** shift each 64-bit block by 4 bytes to the right and, then, shift each 32-bit block by 4 bits to the left. As a result, the numbers are distributed over 32 least significant bits of the register YMM0 as follows: YMM0[0 : 3] = 0, YMM0[4 : 7] = t_4, YMM0[8 : 11] = 0, YMM0[12 : 15] = t_5, YMM0[16 : 19] = 0, YMM0[20 : 23] = t_6, YMM0[24 : 27] = 0, and YMM0[28 : 31] = t_7, Upon applying the XOR operation to the registers YMM0 and YMM4, all eight 4-bit numbers are contained in 32 least significant bits of the register YMM4. It is this 32-bit number that is the output of the GM55 generator.

5. AVX ALGORITHM FOR THE LFSR113 THAT SIMULTANEOUSLY GENERATES TWO RANDOM NUMBER STREAMS

Table 4 shows the implementations of the AVX (left column) and ANSI C (right column) algorithms for the LFSR113 generator. The first part of the AVX algorithm does the same thing as the ANSI C algorithm, but all its operations are applied directly to a vector of eight numbers. The first four numbers of this vector belong to the first random number stream; the other four numbers, to the second stream.

The second random number stream is initialized by skipping 2^{110} numbers in the output sequence of the first stream. The skipping algorithm for the LFSR113 generator was developed by authors of this paper and presented in [31, 37]. Table 3 does not include the corresponding initialization procedure because it is written in C without using AVX vectorization.

Upon calculating new states for both the generators, the instruction **vmovaps** writes thereof into the state variable. The output numbers of the given two streams are $z1 \oplus z2 \oplus z3 \oplus z4$ and $z5 \oplus z6 \oplus z7 \oplus z8$. To evaluate these numbers, the XOR operation is applied to the registers YMM1 and YMM2, which contain the numbers $z1, z2, z3, z4, z5, z6, z7, z8$ and $z3, z4, z1, z2, z7, z8, z5, z6$, respectively. As a result, the register YMM3 contains $z1 \oplus z3$, $z2 \oplus z4$, $z3 \oplus z1$, $z_4 \oplus z2$, $z5 \oplus z7$, $z6 \oplus z8$, $z7 \oplus z5$, $z6 \oplus z8$, $z7 \oplus z5$, and $z8 \oplus z6$, while the register YMM2 contains $z4 \oplus z2$, $z4 \oplus z2$, $z4 \oplus z2$, $z4 \oplus z2$, $z8 \oplus z6$, $z8 \oplus z6$, $z8 \oplus z6$, $z8 \oplus z6$. Then, the XOR operation is applied to the registers YMM3 and YMM2. As a result, the register YMM3 contains $z1 \oplus z2 \oplus z3 \oplus z4$, 0, $z1 \oplus z2 \oplus z3 \oplus z4$, 0, $z5 \oplus z6 \oplus z7 \oplus z8$, 0, $z5 \oplus z6 \oplus z7 \oplus z8$, 0. To extract the output values of both the generators, the algorithm uses instructions **vpextrd**.

6. AVX ALGORITHM FOR THE MRG32K3A

Effective use of the AVX2 technology requires modification of the original algorithm described in Section 1. This modified algorithm should yield the same output sequence but also enable an effective AVX2 vectorization.

For this purpose, we derived new recurrence relations that make it possible to independently evaluate each eight numbers of the sequences $\{x_i\}$, $\{y_i\}$, and $\{u_i\}$, which are given by relations (6), (7), and (8), respectively. First, for each n, x_{n+i} is expressed in terms of x_i, x_{i+1}, x_{i+2} and y_{n+i} is expressed in terms of y_i, y_{i+1}, y_{i+2} $(i = 0,1,2,\ldots)$. For each n, there are coefficients $e_0, e_1, e_2, f_0, f_1, f_2$ such that the following relations hold for all $i = 0,1,2,\ldots$: $x_{n+i} = e_0 x_i + e_1 x_{i+1} + e_2 x_{i+2}$, $y_{n+i} = f_0 y_i + f_1 y_{i+1} + f_2 y_{i+2}$.

Particularly,

$$x_{i+8} = (g_0 x_i + g_1 x_{i+1} + g_2 x_{i+2}) \bmod m_1,$$

$$y_{i+8} = (u_0 y_i + u_1 y_{i+1} + u_2 y_{i+2}) \bmod m_2,$$

$$x_{i+12} = (h_0 x_i + h_1 x_{i+1} + h_2 x_{i+2}) \bmod m_1,$$

Table 4. Implementations of the random number generation algorithm for the LFSR113

AVX2 implementation

```
typedef struct{
  unsigned  z [8]
_attribute_     ((aligned(32)));
} lfsrll3_state;

unsigned  Ifsrll3_consts [32]
_attribute_  ((aligned(32)))={4294967294U,
4294967288U,4294967280U,4294967168U,
4294967294U,4294967288U,4294967280U,4294967168U,
6,2,13,3, 6,2,13,3, 13,27,21,12, 13,27,21,12,
18,2,7,13, 18,2,7,13};

void
lfsrll3_avx_generate_two_(lfsrll3_state*
state,  unsigned  * outputl,  unsigned
*output2){
  asm  volatile(
    "vmovaps   (%2),%%ymm1\n"\
    "vmovaps   (%3),%%ymm5\n"\
    "vandps %%ymml, %%ymm5,%%ymm2\n "\
    "vpsllvd 96(%3),%%ymm2,%%ymm2\n"\
    "vpsllvd 32(%3),%%ymm1,%%ymm4\n"\
    "vxorps %%ymm4,%%ymm1, %%ymm4\n"\
    "vpsrlvd 64(%3),%%ymm4,%%ymm4\n"\
    "vxorps %%ymm4,%%ymm2,%%ymml\n "\
    "vmovaps %%ymm1, (7,2)\n"\
    "vpshufd $78,%%ymm1,%%ymm2\n"\
    "vxorps %%ymm2,%%ymml, %%ymm3\n"\
    "vpshufd $255,%%ymm3,%%ymm2\n"\
    "vxorps %%ymm2,%%ymm3,%%ymm3\n"\
    "vpextrd $0,%%xmm3,%0\n"\
    "vextractf128 $l,%%ymm3,   %%xmm3\n"\
    "vpextrd $0,%%xmm3,%l\n"\
"":"=&r"(*outputl),"=&r"(*output2):"r"
(state->z),"r"(Ifsrll3_consts));
}
```

ANSI C implementation

```
unsigned  zl,z2,z3,z4;

unsigned int  Ifsrll3_ansi_generate_(){
  unsigned  b;
  b = ((zl <<  6) ^ zl) >> 13;
  zl = ((zl  & 4294967294U) << 18)^b;
  b = ((z2 << 2) ^  z2) >> 27;
  z2 = ((z2 & 4294967288U) << 2)^b;
  b = ((z3 << 13) ^ z3) >> 21;
  z3 = ((z3  & 4294967280U) << 7)^b;
  b = ((z4 <<  3) ^ z4) >> 12;
  z4 = ((z4 & 4294967168U)  << 13)^b;
  return   (zl ^ z2 ^ z3 ^ z4);
}
```

$$y_{i+12} = (v_0 y_i + v_1 y_{i+1} + v_2 y_{i+2}) \bmod m_2.$$

Using the recurrence relations of the MRG32K3A generator (see Section 1), the coefficients g_0, g_1, g_2, h_0, h_1, h_2, u_0, u_1, u_2, v_0, v_1, v_2 are expressed in terms of the coefficients a, b, c, d of the original algorithm:

$$g_0 = (2a^2 b) \bmod m_1,$$

$$g_1 = (3ab^2) \bmod m_1,$$

$$g_2 = (a^2 + b^3) \bmod m_1,$$

$$h_0 = (a^4 + 4a^2 b^3) \bmod m_1,$$

$$h_1 = (4a^3 b + b^4) \bmod m_1,$$

$$h_2 = (6a^2 b^2 + b^5) \bmod m_1,$$

$$u_0 = (3c^2 d^2 + cd^5) \bmod m_2,$$

$$u_1 = (2c^2 d + cd^4) \bmod m_2,$$

$$u_2 = (c^2 + 4cd^3 + d^6) \bmod m_2,$$

$$v_0 = (c^4 + 10c^3 d^3 + 7c^2 d^6 + cd^9) \bmod m_2,$$

$$v_1 = (c^4 + 10c^3 d^3 + 7c^2 d^6 + cd^9) \bmod m_2,$$

$$v_2 = (4c^3 d + 15c^2 d^4 3^7 + d^{10}) \bmod m_2.$$

The generator state contains the numbers $x_0, x_1, \ldots, x_7$, $y_0, y_1, \ldots, y_7$. The structure **mrg32k3a_state** includes

Table 5. Implementations of the random number calculation procedure for the MRG32K3A

```
unsigned mrg32k3a_avx_consts[128]
_attribute_     ((aligned(32)))  = {
g0,0,g0,0,g0,0,g0,0, g1,0,g1,0,g1,0,g1,0,
g2,0,g2,0,g2,0,g2,0, h0,0,h0,0,h0,0,h0,0,
h1,0,h1,0,h1,0,h1,0, h2,0,h2,0,h2,0,h2,0,
u0,0,u0,0,u0,0,u0,0, u1,0,u1,0,u1,0,u1,0,
u2,0,u2,0,u2,0,u2,0, v0,0,v0,0,v0,0,v0,0,
v1,0,v1,0,v1,0,v1,0, v2,0,v2,0,v2,0,v2,0,
m1,0,m1,0,m1,0,m1,0, m2,0,m2,0,m2,0,m2,0,
m1-1,0,m1-1,0,m1-1,0,m1-1,0, m2-1,0,m2-1,
0,m2-1,0,m2-1,0   };

typedef  struct{
unsigned  s [33]
_attribute_     ((aligned(32)));
unsigned arrr [8]
_attribute_     ((aligned(32)));
} mrg32k3a_state;

void mrg32k3a.avx_genR.and8(mrg32k3a_state*
state){
  asm volatile(
/* Calculating the output value: */
     "vmovaps (%1),%%ymm0\n"\
     "vmovaps 32(%1),%%ymm6\n"\
     "vpsubq 64(%1),%%ymm0,%%ymml\n"\
     "vpsubq 96(%1),%%ymm6, %%ymm3\n"\
     "vpsrad $31,%%ymm1, %%ymm4\n"\
     "vpsrad $31,%%ymm3,%%ymm7\n"\
     "vpsrlq $32,%%ymm4,%%ymm4\n"\
     "vpsrlq $32,%%ymm7,%%ymm7\n"\
     "vpand 384(%2),%%ymm4,%%ymm4\n"\
     "vpand 384(%2),%%ymm7,%%ymm7\n"\
     "vpaddq %%ymm4,%%ymml,%%ymm1\n"\
     "vpaddq %%ymm7, %%ymm3,%%ymm3\n"\
     "vshufps $136,%%ymm3,%%ymml,%%ymm1\n"\
     "vpermpd $216,%%ymml,%%ymml\n"\
     "vmovaps %%ymml, (7»0)\n"\
/* Evaluating the recurrence relations: */
     "vmovups 8(%1),%%ymm1\n"\
     "vmovups 16(%1),%%ymm2\n"\
     "vpmuludq (%2),%%ymm0,%%ymm3\n"\
     "vpmuludq 32(%2),%%ymm1,%%ymm4\n"\
     "vpmuludq 64(%2),%%ymm2,%%ymm5\n"\
     "vpsrlq $32,%%ymm4,%%ymm6\n"\
     "vpmuludq 384(%2),%%ymm6,%%ymm6\n"\
     "vpsubq %%ymm6,%%ymm4,%%ymm4\n"\
     "vpaddq %%ymm3,%%ymm5,%%ymm3\n"\
     "vpsrlq $32,%%ymm3,%%ymm6\n"\
     "vpmuludq  384(%2),%%ymm6,%%ymm6\n''\
     "vpsubq %%ymm6,%%ymm3,%%ymm3\n"\
     "vpaddq %%ymm3,%%ymm4,%%ymm3\n"\
     "vpsrlq $32,%%ymm3,%%ymm6\n"\
     "vpmuludq 384(%2),%%ymm6,%%ymm6\n"\
```

```
const long long
addl=9007203111403311U;
const long long
add2=9007202867859652U;

const   long long ml = 4294967087U;
const   long long m2 = 4294944443U;
const   long long a = -810728;
const   long long b = 1403580;
const   long long c = -1370589;
const   long long d = 527612;
long long x0, x1, x2, y0, y1, y2;

unsigned mrg32k3a (){
  long k; long long p1, p2;
  p1 = (add1 + b*x1 + a*x0) % m1;
  x0 = x1; x1 = x2; x2 = p1;
  p2 = (add2 + d*y2 + c*y0) % m2;
  y0 = y1; y1 = y2; y2 = p2;
  if(p1 <= p2) return (p1 - p2 + m1);
     else return (p1 - p2);
}
```

Table 5. (Contd.)

```
    "vpsubq %%ymm6,%%ymm3,%%ymm3\n"\
    "vpsrlq $32,%%ymm3,%%ymm6\n"\
    "vpmuludq 384(%2),%%ymm6,%%ymm6\n"\
    "vpsubq %%ymm6,%%ymm3,%%ymm3\n"\
    "vmovaps %%ymm3,   (%1)\n"\
/*      [...]                          */
    ""::"r"(state->arrr), "r"(state>s),
"r"(mrg32k3a_avx_consts));
}
unsigned int mrg32k3a_avx_generate_(mrg32k3a_state*
state){
  state->s [32]- -;
  if(state->s [32]==0){
    mrg32k3a_avx_genRand8(state);
    state->s [32]=8;
  }
  return state->arrr[8-state->s [32]];
}
```

an array s, which stores these numbers in the following order: , 0, $x_3, 0, x_4, 0$, $x_5, 0, x_6, 0$, $x_7, 0, y_0, 0, y_1$, $0, y_2, 0, y_3, 0, y_4, 0, y_5, 0, y_6, 0, y_7, 0$ (see the left column in Table 5). Zeros are placed among the numbers for convenient use of the AVX instructions.

Table 5 shows the implementations of the procedure for calculating the output number $u_i = (x_i - y_i) \bmod m_1$. Upon applying the instruction **vpsubq**, the register YMM1 contains the 64-bit values $x_0 - y_0$, $x_1 - y_1$, $x_2 - y_2$, $x_3 - y_3$, while the register YMM3 contains the 64-bit values $x_4 - y_4$, $x_5 - y_5$, $x_6 - y_6$, $x_7 - y_7$. If the difference is negative, then m_1 needs to be added. The information about the sign is stored in the most significant bit of each 64-bit word. If the difference is positive, then the instruction **vpsrad** writes zero bits for the highest of two 32-bit words; if the difference is negative, it writes unit bits. The instruction **vpsrlq** carries over the data from all 32-bit high words into low words. If the low word contains unit bits, then, upon applying the instruction **vpand**, this word is replaced by m_1; the numbers obtained are then added to the YMM1 by using the 64-bit instruction **vpaddq**. Thus, the register YMM1 contains the 32-bit values $u_0, 0, u_1, 0$, $u_2, 0, u_3, 0$, while the register YMM3 contains $u_4, 0, u_5, 0$, $u_6, 0, u_7, 0$. Upon applying the instruction **vshufps**, the register YMM1 contains the numbers u_0, u_1, u_4, u_5, u_2, u_3, u_6, u_7. Finally, using the instruction **vpermpd**, we swap the numbers and obtain the output values u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7.

Another important part of the algorithm is the procedure for calculating a new state of the generator by using recurrence relations (6) and (7). The instruction **vpmuludq** multiplies the first, third, fifth, and seventh 32-bit numbers of the first operand by the corresponding numbers of the second operand and, then, writes the 64-bit results into the third operand. First, the numbers are placed into the registers YMM0, YMM1, and YMM2, so the YMM0 contains $x_0, 0, x_1, 0, x_2, 0, x_3, 0$; the YMM1 contains $x_1, 0, x_2, 0$, $x_3, 0, x_4, 0$; and the YMM2 contains $x_2, 0, x_3, 0$, $x_4, 0, x_5, 0$. Next, the register YMM0 is multiplied by $g_0, 0, g_0$, $0, g_0, 0, g_0$; the register YMM1 is multiplied by $g_1, 0, g_1, 0$, $g_1, 0, g_1, 0$; and the register YMM2 is multiplied by $g_2, 0, g_2, 0, g_2, 0, g_2, 0$. Then, the operation $z \Leftarrow z - m_1 \times [z/2^{32}]$ is used to (considerably) reduce the value obtained without affecting the remainder on dividing by m_1. This allows the multiplication results to be added up without exceeding the limits of the 64-bit number (the algorithm takes into account the values of the coefficients: in certain cases, some operations can be skipped without exceeding the limits). There is no special AVX instruction that would simultaneously take remainders on dividing by m, so the operation $z \Leftarrow z - m \times [z/2^{32}]$ is carried out several times to ensure the correctness of calculations. As a result, we obtain the numbers x_8, x_9, x_{10}, x_{11}, each of which does not exceed 2^{32}. The numbers $x_{12}, x_{13}, x_{14}, x_{15}$ and y_8, y_9, y_{10}, y_{11}, $y_{12}, y_{13}, y_{14}, y_{15}$ are evaluated similarly.

7. AVX ALGORITHM FOR THE MT19937

Table 6 shows the AVX implementation of the random number generation algorithm for the MT19937 generator. The function **mt19937_avx_do16all** evaluates 16 recurrence relations of the form (1). The argument i takes values from 0 to 623 that are divisible by 16. The MT19937 algorithm uses the numbers in the

Table 6. Implementations of the random number generation algorithm for the MT19937

AVX2 implementation

```
unsigned mtl9937_avx_consts[56]
 _attribute_ ((aligned(32)))={
  0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,
  upper_mask, upper_mask, upper_mask,
upper_mask, upper_mask, upper_mask,
upper_mask, upper_mask,
  lower_mask, lower_mask, lower_mask,
lower_mask, lower_mask, lower_mask,
lower_mask, lower_mask,
  andAmask, andAmask, andAmask, andAmask,
andAmask, andAmask, andAmask, andAmask,
b_mask, b_mask, b_mask, b_mask, b_mask,
b_mask, b_mask, b_mask,
  c_mask, c_mask, c_mask, c_mask, c_mask,
c_mask, c_mask, c_mask
};
typedef struct{
  unsigned mt_aligned[3*mtl9937_N+9]
 _attribute_ ((aligned(32)));
  unsigned out [3*mtl9937_N+9]
 _attribute_((aligned(32)));
  unsigned *mt;   int  mti;
} mtl9937_state;

void   mtl9937_avx_dol6all(int
i,mtl9937_state* state){
 asm  volatile(
   "vmovaps 2500(%0),%%ymm0\n" \
   "vmovaps 2532(%0),%%ymm4\n" \
   "vmovaps 4(%0),%%ymm1\n" \
   "vmovaps 36(%0),%%ymm5\n" \
   "vandps 64(%2),%%ymm0,%%ymm0\n" \
   "vandps 64(%2),%%ymm4,%%ymm4\n" \
   "vandps 96(%2),%%ymm1,%%ymm1\n" \
   "vandps 96(%2),%%ymm5,%%ymm5\n" \
   "vorps %%ymm0,%%ymm1,%%ymm1\n" \
   "vorps %%ymm4,%%ymm5,%%ymm5\n" \
   "vmovaps %%ymm1,%%ymm0\n" \
   "vmovaps %%ymm5,%%ymm4\n" \
   "vpsrld $l,%%ymml,%%ymm1\n" \
   "vpsrld $l,%%ymm5,%%ymm5\n" \
   "vandps 32(%2),%%ymm0,%%ymm0\n" \
   "vandps 32(%2),%%ymm4,%%ymm4\n" \
   "vpcmpgtd (%2),%%ymm0,%%ymm0\n" \
   "vpcmpgtd (%2),%%ymm4,%%ymm4\n" \
   "vandps 128(%2),%%ymm0,%%ymm0\n" \
   "vandps 128(%2),%%ymm4,%%ymm4\n" \
   "vxorps %%ymm1//,%ymm0,%%ymm0\n'' \
   "vxorps %%ymm5//,%ymm4,%%ymm4\n'' \
   "vmovups 1588(%0),%%ymm1\n" \
   "vmovups 1620(%0),%%ymm5\n" \
   "cmpl $223,%3\n" \
   "jl MyL1%=\n" \
```

ANSI C implementation

```
#define N 624
#define M 397
#define L 227
#define NPLUSL 851
#define MATRIX_A 0x9908b0dfUL
#define UPPER_MASK 0x80000000UL
#define LOWER_MASK 0x7fffffffUL
#define b_mask 0x9d2c5680UL
#define c_mask 0xefc60000UL



unsigned
mtl9937_ansi_generate_(mtl9937_state*state)
{
  unsigned y;
  if (state->mti >= N) {
    int kk;
    for (kk=0;kk<N-M;kk++) {

    y = (state->mt[kk]&UPPER\_MASK) I
(state->mt[kk+l]&LOWER_MASK);

    state->mt[kk] = state->mt[kk+M]^(y >> 1)
^((y & 0x1UL)?MATRIX_A:0);

    }

    for (;kk<N-l;kk++) {

    y = (state->mt[kk]&UPPER_MASK) I
(state->mt[kk+l]&LOWER_MASK);

    state->mt [kk] = state->mt [kk+(M-N)]^(y>>l)
^((y & 0x1UL)?MATRIX_A:0);

    }

    y = (state->mt[N-l]$\&$UPPER\_MASK) I
(state->mt[0]$\&$LOWER\_MASK);

    state->mt[N-l] = state->mt[M-l]^(y >> 1)^
((y & 0x1UL)?MATRIX_A:0);

    state->mti = 0;
  }

  y = state->mt[state->mti++];

  y^= (y >> 11);
```

Table 6. (Contd.)

AVX2 implementation	ANSI C implementation

`"cmpl $227,%3\n" \`
`"jl MyL2%=\n" \`
`"vmovups -908(%0),%%ymm1\n" \`
`"MyL2%=: \n" \`
`"vmovaps -892(%0),%%ymm3\n" \ "`
`"vinsertil28 $1,-892(%0) ,%%ymm1,%%ymm1\n"\`
`"vmovups -876(%0),%%ymm5\n" \`
`"MyLl%=: \n" \`
`"vxorps %%ymml,%%ymm0,%%ymm0\n" \`
`"vxorps %%ymm5,%%ymm4,%%ymm4\n" \`
`"vmovups %%ymm0, (%0) \n" \`
`"vmovups %%ymm4,32(%0)\n" \`
`"vmovaps %%ymm0,2500(%0) \n" \`
`"vmovaps %%ymm4,2532 (%0) \n" \`
`"vpsrld $11,%%ymm0,%%ymm2\n" \`
`"vpsrld $11,%%ymm4,%%,ymm3\n" \`
`"vxorps %%ymm0,%%ymm2,%%ymm0\n" \`
`"vxorps %%ymm4,%%ymm3,%%ymm4\n" \`
`"vpslld $7,%%ymm0,%%ymm2\n" \`
`"vpslld $7,%%ymm4,%%ymm3\n" \`
`"vandps 160(%2),%%ymm2,%%ymm2\n" \`
`"vandps 160(%2),%%ymm3,%%ymm3\n" \`
`"vxorps %%ymm0,%%ymm2,%%ymm0\n" \`
`"vxorps %%ymm4,%%ymm3,%%ymm4\n" \`
`"vpslld $15,%%ymm0,%%ymm2\n" \`
`"vpslld $15,%%ymm4,%%,ymm3\n" \`
`"vandps 192(%2),%%ymm2,%%ymm2\n" \`
`"vandps 192(%2),%%ymm3,%%ymm3\n" \`
`"vxorps %%ymm0,%%ymm2,%%ymm0\n" \`
`"vxorps %%ymm4,%%ymm3,%%ymm4\n" \`
`"vpsrld $18,%%ymm0,%%ymm2\n" \`
`"vpsrld $18,%%ymm4,%%,ymm3\n" \`
`"vxorps %%ymm0,%%ymm2,%%ymm0\n" \`
`"vxorps %%ymm4,%%ymm3,%%ymm4\n" \`
`"vmovaps %%ymm0, (%1) \n" \`
`"vmovaps %%ymm4,32(%1)\n" \`
`"":"r"(state->mt+i),"r"(state->out+i),`
`"r"(mtl9937_avx_consts),"r"(i));`
`}`
 	
`unsigned int mtl9937_avx_generate_(mtl9937_state* state){`
`int I;`
`if (state->mti>=mtl9937_N) {`
`mtl9937_avx_dol6all(0,state);`
`asm(`
`"movl 2500(%0),%%eax;"\`
`"movl %%eax, 2496 (%0)"\`
`::"r"(state->mt):'%eax");`
`for (i=16;i<mtl9937_N;i+=16)`
`mtl9937_avx_dol6all(I,state);`
`state->mti=0;`
`}`
`return state->out[state->mti++];`
`}`
	`y^= (y >> 7) & b_mask;`
	`y^= (y << 15) & c_mask;`
	`y^= (y >> 18);`
	
	`return y;`
	`}`

Table 7. Performance of the random number generators (processor: Intel Xeon E5-2650v3 (2.3 GHz); compiler: gcc; optimization: –03)

Generator	ANSI C: (Gbit/sec)	SSE (Gbit/sec)	AVX (Gbit/sec)	Relative speed-up-Speed(SSE)/Speed (ANSI C)	Relative speed-up-Speed(AVX)/Speed (SSE)
GM19	0.27	1.43	3.12	5.30	2.17
GM29	0.32	1.86	3.11	5.84	1.67
GM31	0.31	1.29	2.17	4.14	1.69
GM55	0.93	2.30	3.27	2.47	1.42
GM61	0.14	0.48	0.89	3.39	1.84
GQ58.1	0.27	0.63	1.12	2.30	1.79
GQ58.3	0.58	1.31	2.15	2.25	1.65
GQ58.4	0.98	1.97	3.10	2.02	1.57
LFSR113	5.98	3.83	12.61	0.64	3.29
MRG32K3A	2.84	5.21	7.64	1.83	1.46
MT19937	7.43	9.84	12.23	1.32	1.24

array with the period 624, i.e., recurrence relation (1) implies that $x_{i+624} = x_i$ for all i's. The AVX algorithm (see the left column in Table 6) writes the numbers into the array **mt** and reads them therefrom with the period 625. Thus, $x_{i+625} = x_i$ if $ib5$ does not exceed a certain threshold. This provides quick simultaneous access to both the number sets $x_i,\ldots,x_{i+15}$ and $x_{i+1}\ldots,x_{i+16}$.

The algorithm places the numbers $x_i, x_{i+1}\, dots, x_{i+15}$ into the registers YMM0 and YMM4 and places the numbers $x_{i+1}, x_{i+2},\ldots,x_{i+16}$ into the registers YMM1 and YMM5. Upon applying the instructions **vandps**, the YMM0 and YMM4 contain $x_i^u, x_{i+1}^u,\ldots,x_{i+15}^u$, while YMM1 and YMM5 contain $x_{i+1}^l, x_{i+2}^l,\ldots,x_{i+16}^l$. Upon applying the OR operation, the registers contain $(x_i^u|x_{i+1}^l)$, $(x_{i+1}^u|x_{i+2}^l)$, ..., $(x_{i+15}^u|x_{i+16}^l)$. Then, the multiplication by the companion matrix A is carried out. The instruction **vpsrld** simultaneously shifts all 32-bit numbers in YMM1 and YMM5 by one bit to the right. Next, the least significant bits of all 16 numbers in the registers YMM0 and YMM4 are compared with zero. The instructions **vpcmpgtd** and **vandps** write into the registers YMM0 and YMM4 16 numbers each of which is zero if the corresponding number from the set $(x_i^u|x_{i=1}^l)$, $(x_{i+1}^u|x_{i+2}^l)$, ..., $(x_{i+15}^u|x_{i+16}^l)$ is even (if it is odd, then each of these 16 numbers is **a**). Thus, upon applying the operations **vxorps**, the registers YMM0 and YMM4 contain the numbers $(x_i^u|x_{i+1}^l)A$, $(x_{i+1}^u|x_{i+1}^l)A,\ldots,(x_{i+15}^u|x_{i+16}^l)A$.

Next, the numbers $x_{i+397(\mathrm{mod}\,624)}$, $x_{i+398(\mathrm{mod}\,624)}$, ..., and $x_{i+412(\mathrm{mod}\,624)}$ need to be placed into the registers YMM1 and YMM5. For this purpose, the algorithm does the following. If $i \le 208$, then the numbers x_{i+397}, x_{i+398}, and x_{i+412} are written into these registers; if i"4, then the numbers x_{i+397}, x_{i+398}, x_{i+399}, x_{i+400}, x_{i-223}, $x_{i-222},\ldots,$ x_{i-212} are written; and if $i \ge 240$, then the numbers x_{i-227}, $x_{i-226},\ldots,$ x_{i-212} are written. These numbers are compared with 223 and 227. Since i is divisible by 16, $i < 223$ implies that $i \le 208$; $223 \le i < 227$ implies that $i = 224$; and $i \ge 227$ implies that $i \ge 240$. Then, the XOR operation is performed to evaluate the right-hand side of expression (1) for $k = i, i+1,\ldots,i+15$. The result is written into $x_i, x_{i+1},\ldots,x_{i+15}$ and into x_{i+625}, $x_{i+626},\ldots,$ x_{i+640}.

Note that, since i takes values from 0 to 623 that are divisible by 16, the size of the array **mt** should not be less than 1249. To guarantee that $x_{624} = x_0$ (which is nessesary because x_{624} is used in the case where $i = 224$), the function **mt19937_avx_generate_** does the corresponding copying. The pointer of **mt** is initialized by the expression **mt=mt_aligned+7**. In this case, **mt** is shifted by 28 bytes from the 32-byte boundary, which allows one to use the faster instructions **vmovaps** (instead of **vmovups**).

Then, the multiplication by the matrix T is carried out by successive application of formulas (2)–(5). The instruction **vpsrld** performs the right shift by 11 bits; particularly, the numbers $x_i \gg u$, $x_{i+1} \gg u,\ldots,$ and $x_{i+7} \gg u$ are placed into the register YMM2. Next, the XOR operation is carried out, and the numbers $x_i \oplus (x_i \gg u)$, $x_{i+1} \oplus (x_{i+1} \gg u),\ldots,$ and $x_{i+7} \oplus (x_{i+7} \gg u)$ are written into YMM0 (i.e., formula (2) is applied). Then, the left shift by 7 bits and the AND operation using the mask **b** are performed. After the XOR operation, the register YMM0 contains $y_i \oplus ((y_i \ll s)\, AND\, \mathbf{b})$, $y_{i+1} \oplus ((y_{i+1} \ll s)\, AND\, \mathbf{b})$, ...,

Table 8. Performance of the random number generators (processor: Intel Core i7-4790K (4 GHz); compiler: gcc; optimization: –03)

Generator	ANSI C: (Gbit/sec)	SSE (Gbit/sec)	AVX (Gbit/sec)	Relative speed-up-Speed(SSE)/Speed (ANSI C)	Relative speed-up-Speed(AVX)/Speed (SSE)
GM19	0.45	2.05	4.52	4.55	2.21
GM29	0.51	2.69	4.69	5.27	1.74
GM31	0.53	1.92	3.32	3.63	1.73
GM55	1.36	3.57	5.16	2.62	1.45
GM61	0.21	0.75	1.32	3.5	1.76
GQ58.1	0.44	0.93	1.67	2.12	1.8
GQ58.3	0.85	2.01	3.3	2.37	1.63
GQ58.4	1.38	2.98	4.28	2.16	1.44
LFSR113	10.7	5.6	18.2	0.52	3.25
MRG32K3A	4.2	8.79	12.4	2.09	1.41
MT19937	8.85	14.5	17.8	1.64	1.23

Table 9. Performance of the random number generators (processor: Intel Xeon E5-2650v3 (2.3 GHz); compiler: gcc; optimization: –00)

Generator	ANSI C (Gbit/sec)	SSE (Gbit/sec)	AVX (Gbit/sec)	Relative speed-up-Speed(SSE)/Speed (ANSI C)	Relative speed-up-Speedi(AVX)/Speed(SSE)
GM19	0.06	1.22	2.38	20.37	1.95
GM29	0.07	1.53	2.54	22.73	1.66
GM31	0.04	1.10	1.78	26.26	1.62
GM55	0.37	2.02	2.90	5.52	1.44
GM61	0.03	0.47	0.83	17.00	1.77
GQ58.1	0.06	0.58	0.98	10.17	1.69
GQ58.3	0.23	1.20	1.87	5.27	1.56
GQ58.4	0.36	1.75	2.54	4.91	1.45
LFSR113	3.03	3.72	9.94	1.23	2.67
MRG32K3A	0.84	3.84	4.48	4.58	1.17
MT19937	1.70	6.61	7.22	3.89	1.09

and $y_{i+7} \oplus ((y_{i+7} \ll s)\, AND\, \mathbf{b})$; i.e., formula (3) is applied. The other registers are filled in similarly, and the remaining operations for evaluating the right-hand sides of expressions (2)–(5) are carried out. The evaluated output numbers $z_i, z_{i+1}, \ldots, z_{i+15}$ of the generator are written into the array **out**.

8. SPEED OF PSEUDORANDOM NUMBER GENERATION

Tables 7, 8, 9, and 10 demonstrate the performance of the generators considered above on the Intel Xeon E5-2650v3 and Intel Core i7-4790K processors, which are widely used in modern servers and workstations, respectively. In tests, the gcc compiler with the optimization levels –00 and –03 was used. It can be seen that AVX vectorization significantly improves the performance of the generators. The AVX implementations yield exactly the same pseudorandom sequences as the original algorithms, while being up to 40 times faster than the ANSI C implementations.

CONCLUSIONS

In this paper, modern random number generation algorithms have been analyzed, and the use of AVX vectorization for calculation speed-up has been considered. The approach described in this paper was employed in the development of the RNGAVXLIB library [35].

Table 10. Performance of the random number generators (processor: Intel Core i7-4790K (4 GHz); compiler: gcc; optimization: –00)

Generator	ANSI C: (Gbit/sec)	SSE (Gbit/sec)	AVX (Gbit/sec)	Relative speed-up-Speed(SSE)/Speed (ANSI C)	Relative speed-up-Speed(AVX)/Speed (SSE)
GM19	0.09	1.82	3.50	20.20	1.92
GM29	0.10	2.34	3.71	23.57	1.59
GM31	0.06	1.61	2.74	26.18	1.70
GM55	0.54	2.97	4.23	5.53	1.42
GM61	0.04	0.71	1.24	17.52	1.73
GQ58.1	0.08	0.88	1.52	10.44	1.73
GQ58.3	0.34	1.82	2.83	5.39	1.56
GQ58.4	0.53	2.50	3.66	4.73	1.46
LFSR113	4.49	5.25	15.09	1.17	2.87
MRG32K3A	1.23	5.34	6.54	4.34	1.22
MT19937	2.55	9.12	8.65	3.58	0.95

ACKNOWLEDGMENTS

This work was supported by the Russian Science Foundation, project no. 14-21-00158.

REFERENCES

1. Beach, K.S.D., Lee, P.A., and Monthoux, P., Field-induced antiferromagnetism in the Kondo insulator, *Phys. Rev. Lett.,* 2004, vol. **92**, no. 2.
2. Binder, K. and Heerman, D.W., *Monte Carlo Simulation in Statistical Physics: An Introduction,* Springer, 2010, 5th ed.
3. Landau, D.P. and Binder, K., *A Guide to Monte–Carlo Simulations in Statistical Physics,* Cambridge Univ. Press, 2000.
4. Pieper, S.C. and Wiring, R.B., Quantum Monte–Carlo calculations of light nuclei, *Ann. Rev. Nucl. Part. Sci.,* 2001, vol. **51**, pp. 53–90.
5. Luchow, A. and Anderson, J.B., Monte–Carlo methods in electronic structures for large molecules, *Ann. Rev. Phys. Chem.,* 2000, vol. **51**, pp. 501–526.
6. Bizzarri, A.R., Neutron scattering and molecular dynamics simulation: A conjugate approach to investigate the dynamics of electron transfer proteins, *J. Phys.: Cond. Mat.,* 2004, vol. **16**, no. 4, pp. R83–R110.
7. Lehmer, D.H., Mathematical methods in large-scale computing units, *Proc. Second Symp. on Large-Scale Digital Calculating Machinery,* Cambridge, 1951, pp. 141–146.
8. Coveyou, R.R. and MacPherson, R.D., Fourier analysis of uniform random number generators, *J. Assoc. Comput. Mach.,* 1967, vol. **14**, pp. 100–119.
9. Marsaglia, G., Random numbers fall mainly in the planes, *Proc. Nat. Acad. Sci. USA,* 1968, vol. **61**, pp. 25–28.
10. Golomb, S.W., *Shift Register Sequences,* San Francisco: Holden-Day, 1967.
11. Shchur, L.N., On the quality of random number generators with taps, *Comput. Phys. Commun.,* 1999, nos. 121–122, pp. 83–85.
12. Ferrenberg, A.M., Landau, D.P., and Wong, Y.J., Monte–Carlo simulations: Hidden errors from "good" random number generators, *Phys. Rev. Lett.,* 1992, vol. **69**, no. 23.
13. Selke, W., Talapov, A.L., and Shchur, L.N., Cluster-flipping Monte–Carlo algorithm and correlation in "good" random number generators, *JETP Lett.,* 1993, vol. **58**, p. 665.
14. Grassberger, P., On correlations in "good" random number generators, *Phys. Lett. A,* 1993, vol. **181**, pp. 43–46.
15. Vattulainen, I., Ala-Nissila, T., and Kankaala, K., Physical tests for random numbers in simulations, *Phys. Rev. Lett.,* 1994, vol. **73**, pp. 2513–2516.
16. Schmid, F. and Wilding, N.B., Errors in Monte–Carlo simulations using shift register random number generators, *Int. J. Modern Phys. C,* 1995, vol. **6**, pp. 781–797.
17. Shchur, L.N., Heringa, J.R., and Bloete, H.W.J., Simulations of a directed random-walk model: The effect of pseudo-random-number-correlations, *Physica A,* 1997, vol. **241**, pp. 579–592.
18. Shchur, L.N. and Bloete, H.W.J., Cluster Monte–Carlo: Scaling of systematic errors in 2D Ising model, *Phys. Rev. E,* 1997, vol. **55**, pp. R4905–R4908.
19. L'Ecuyer, P., Random number generation, in *Handbook on Simulation,* Banks, J., Ed., Wiley, 1998, pp. 93–137.
20. L'Ecuyer, P. and Simard, R., TestU01: A C library for empirical testing of random number generators, *ACM TOMS,* 2007, vol. 33, no. 4.
21. Matsumoto, M. and Tishimura, T., Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Trans. Mod. Comp. Simul,* 1998, vol. **8**, no. 1, pp. 3–30.

22. L'Ecuyer, P., Good parameter sets for combined multiple recursive random number generators, *Oper. Res,* 1999, vol. **47**, no. 1, pp. 159–164.
23. L'Ecuyer, P., Maximally equidistributed combined Tausworthe generators, *Math. Comp.,* 1996, vol. **65**, no. 213, pp. 203–213.
24. L'Ecuyer, P., Tables of maximally-equidistributed combined LFSR generators, *Math. Comp.,* 1999, vol. **68**, no. 255, pp. 261–269.
25. Barash, L.Yu., Applying dissipative dynamical systems to pseudorandom number generation: Equidistribution property and statistical independence of bits at distances up to logarithm of mesh size, *Europhys. Lett.,* 2011, vol. **95**, pp. 10003–10008.
26. Barash, L. and Shchur, L.N., Periodic orbits of the ensemble of Sinai–Arnold cat maps and pseudorandom number generation, *Phys. Rev. E,* 2006.
27. Barash, L. and Shchur, L.N., RNGSSELIB: Program library for random number generation, SSE2 realization, *Comput. Phys. Commun.,* 2011, vol. **182**, no. 7, pp. 1518–1527.
28. Barash, L.Yu. and Shchur, L.N., Hyperbolic toral automorphisms and pseudorandom number generators: Implementations based on SSE instructions, *Trudy Seminara po vychislitel'nym tekhnologiyam v estestvennykh naukakh, vyp. 1. Vychislitel'naya fizika* (Proc. Workshop on Computational Technologies in Natural Sciences, Series 1: Computational Physics), Nazirov, R.R, Ed., Moscow: Izd. KDU, 2009, pp. 14–29.
29. Barash, L.Yu., Geometric and statistical properties of pseudorandom number generators based on multiple recursive transformations, Springer Proceedings in Mathematics and Statistics, Springer, 2012, vol. **23**, pp. 265–280.
30. Barash, L.Yu. and Shchur, L.N., RNGSSELIB: Program library for random number generation. More generators, parallel streams of random numbers, and Fortran compatibility, *Comput. Phys. Commun.,* 2013, vol. **184**, no. 10, pp. 2367–2369.
31. Barash, L.Yu. and Shchur, L.N., PRAND: GPU accelerated parallel random number generation library. Using most reliable algorithms and applying parallelism of modern GPUs and CPUs, *Comput. Phys. Commun.,* 2014, vol. **185**, no. 4, pp. 1343–1353.
32. L'Ecuyer, P., Uniform Random Number Generation, *Ann. Oper. Res.*, 1994, vol. **53**, pp. 77–120.
33. Salmon, J.K., Moraes, M.A., Dror, R.O., and Show, D.E., Parallel random numbers: As easy as 1, 2, 3, *Proc. Int. Conf. High Performance Computing, Networking, Storage, and Analysis,* New York, 2011, pp. 1–12.
34. Manssen, M., Weigel, M., and Hartmann, A.K., Random number generators for massively parallel simulations on GPU, *Eur. Phys. J.: Spec. Top.,* 2012, vol. **210**, no. 1, pp. 53–71.
35. Guskova, M.S., Barash, L.Yu., and Shchur, L.N., RNGAVXLIB: Program library for random number generation, AVX realization, *Comput. Phys. Commun.,* 2016, vol. **200**, pp. 402–405.
36. Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 2 (2A, 2B, 2C). https://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
37. Barash, L.Yu. and Shchur, L.N., About generation of parallel streams of pseudorandom numbers, *Program. Inzheneriya,* 2013, no. 1, pp. 24–32.
38. Galassi, M., et al., *GNU Scientific Library Reference Manual,* Network Theory Ltd., 2009, 3rd ed.
39. Intel Math Kernel Library Reference Manual, 2007. https://www.intel.com/cd/software/products/emea//rus/358888.htm.
40. Mascagni, M. and Srinivasan, A., Algorithm 806: SPRNG: A scalable library for pseudorandom number generation, *ACM Trans. Math. Software,* 2000, vol. **26**, pp. 436–461.
41. Bauke, H., Tina's random number generator library, 2011. http://www.numbercrunch.de/trng.
42. Barash, L.Yu. and Shchur, L.N., Generating random numbers and parallel random number streams for Monte–Carlo calculations, *Model. Anal. Inf. Sist.,* 2012, vol. **19**, no. 2, pp. 145–162.
43. Bauke, H. and Mertens, S., Random numbers for large-scale distributed Monte–Carlo simulations, *Phys. Rev. E,* 2007.
44. Shchur, L.N. and Butera, P., The RANLUX generator: Resonances in a random walk test, *Int. J. Mod. Phys,* 1998, vol. **4**, no. 9, pp. 607–624.

Translated by Yu. Kornienko

SPELL; OK