An Application of Finite Field: Design and Implementation of 128-bit Instruction-Based Fast Pseudorandom Number Generator

有限体の応用：128ビット命令に基づく高速擬似乱数生成器の設計と実装

指導教官　松本眞教授

広島大学理学研究科数学専攻
斎藤睦夫

2007年2月9日

Abstract

(1) SIMD-oriented Mersenne Twister (SFMT) is a new pseudorandom number generator (PRNG) which uses 128-bit Single Instruction Multiple Data (SIMD) operations. SFMT is designed and implemented on C language with SIMD extensions and also implemented on standard C without SIMD. (2) Properties of SFMT are studied by using finite field theories, and they are shown to be equal or better than Mersenne Twister (MT), which is a widely used PRNG. (3) Generation speed of SFMT is measured on Intel Pentium M, Pentium IV, AMD Athlon 64 and PowerPC G4. It is shown to be about two times faster than MT implemented using SIMD.

1 Introduction

Computer Simulation is one of the most important techniques of modern science. Recently, the scale of simulations is getting larger, and faster pseudorandom number generators (PRNGs) are required. Power of CPUs for usual personal computers is now sufficiently strong for such purposes, and necessity of efficient PRNGs for CPUs on PCs is increasing. One such generator is Mersenne Twister (MT) [10], which is based on a linear recursion modulo 2 over 32-bit words. An implementation MT19937 has the period of $2^{19937} - 1$.

There is an argument that the CPU time consumed for function calls to PRNG routines occupies a large part of the random number generation, and consequently it is not so important to improve the speed of PRNG (cf. [13]). This is not always the case: one can avoid the function calls by (1) inline-expansion and/or (2) generation of pseudorandom numbers in an array at one call.

Our aim of this paper is to design a fast MT-like PRNG (i.e. Linear Feedbacked Shift Register) considering new features of modern CPUs on PCs.

1.1 Linear Feedbacked Shift Register (LFSR) generators

A LFSR method is to generate a sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ of elements $\mathbb{F}_2^w$ by a recursion

$$\mathbf{x}_{i+N} := g(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+N-1}), \tag{1}$$

where $\mathbf{x}_i \in \mathbb{F}_2^w$ and $g : (\mathbb{F}_2^w)^N \to \mathbb{F}_2^w$ is an $\mathbb{F}_2$-linear function (i.e., the multiplication of a $(wN \times w)$-matrix from the right to a wN-dimensional vector) and use it as a pseudorandom w-bit integer sequence. In the implementation, this recursion is computed by using an array `W[0..N-1]` of N integers of w-bit size, by the simultaneous substitutions

$$\mathtt{W[0]} \leftarrow \mathtt{W[1]},\ \mathtt{W[1]} \leftarrow \mathtt{W[2]},\ \ldots, \mathtt{W[N-2]} \leftarrow \mathtt{W[N-1]},\ \mathtt{W[N-1]} \leftarrow g(\mathtt{W[0]}, \ldots, \mathtt{W[N-1]}).$$

The first $N - 1$ substitutions shift the content of the array, hence the name of LFSR. Note that in the implementation we may use an indexing technique to avoid computing these substitutions, see [7, P.28 Algorithm A]. The array

`W[0..N-1]` is called 'state array.' Before starting the generation, we need to set some values to the state array, which is called 'initialization.'

Mersenne Twister (MT) [10] is an example with

$$g(\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}) = (\mathbf{w}_0|\mathbf{w}_1)A + \mathbf{w}_M,$$

where $(\mathbf{w}_0|\mathbf{w}_1)$ denotes a concatenation of $32-r$ MSBs of $\mathbf{w}_0$ and r LSBs of $\mathbf{w}_1$, r is an integer $(1 \leq r \leq 31)$, A is a (32×32)-matrix for which the multiplication $\mathbf{w}A$ is computable by a few bit-operations, and M is an integer $(1 < M < N)$. Its period is $2^{32N-r} - 1$, chosen to be a Mersenne prime. To obtain a better equidistribution property, MT transforms the sequence by a suitably chosen (32×32) matrix T, namely, MT outputs $\mathbf{x}_0T, \mathbf{x}_1T, \mathbf{x}_2T, \ldots$ (called tempering).

An advantage of $\mathbb{F}_2$-linear generators over integer multiplication generators (such as Linear Congruential Generators [7] or Multiple Recursive Generators [8]) was high-speed generation by avoiding multiplications. Another advantage is that the behavior of generated pseudorandom number sequence is theoretically well studied and its dimension of equidistribution can be calculated easily.

1.2 Single Instruction Multiple Data (SIMD)

Single Instruction Multiple Data (SIMD) [16] is a technique employed to achieve data level parallelism. Typically, four 32-bit registers are combined into a 128-bit register, and a single instruction operates on the 128-bit register. There are two types of SIMD instructions. One is to operate four 32-bit registers separately (e.g. addition and subtraction.) The Other is to operate on the 128-bit integer (e.g. 128-bit shift operation.)

SIMD is also called Multimedia Extension because main target applications of SIMD are multimedia applications, which use huge data like image or sound. LFSR use large internal state array, so SIMD is expected to accelerate its generation.

Streaming SIMD Extensions 2 (SSE2) [6, Chapter 4–5] is one of the SIMD instruction sets introduced by Intel. Pentium M, Pentium 4 and later CPUs support SSE2, but Itanium and Itanium 2 do not. AMD Athlon 64, Opteron and Turion 64 also support SSE2. These CPUs have eight 128-bit registers and each register can be divided into 8-bit, 16-bit, 32-bit or 64-bit blocks.

AltiVec [4] is another SIMD instruction-set supported by PowerPC G4 and G5. These CPUs have thirty-two 128-bit registers and each register can be divided into 8-bit, 16-bit or 32-bit blocks.

We tried to design a PRNG which can be implemented efficiently both in SSE2 and AltiVec.

Intel C compiler has an ability to handle SSE2 instructions. GCC C compiler, which is more widely used, also has macros and inline functions to handle SSE2 and AltiVec instructions directly.

2 SIMD-oriented Fast Mersenne Twister

In this article, we propose an MT-like pseudorandom number generator that makes full use of SIMD: SFMT, SIMD-oriented Fast Mersenne Twister. We implemented an SFMT with the period of a multiple of $2^{19937}-1$, named SFMT19937, which has a better equidistribution property than MT. SFMT is faster than MT, even without using SIMD instructions.

SFMT is a LFSR generator based on a recursion over $\mathbb{F}_2^{128}$. We identify the set of bit $\{0,1\}$ with the two element field $\mathbb{F}_2$. This means that every arithmetic operation is done modulo 2. A w-bit integer is identified with a horizontal vector in $\mathbb{F}_2^w$, and + denotes the sum as vectors (i.e., bit-wise exor), not as integers. We consider three cases: w is 32, 64 or 128.

2.1 The recursion of SFMT

We choose g in the recursion (1) as

$$g(\mathbf{w}_0,\ldots,\mathbf{w}_{N-1}) \quad = \quad \mathbf{w}_0 A + \mathbf{w}_M B + \mathbf{w}_{N-2} C + \mathbf{w}_{N-1} D, \tag{2}$$

where $\mathbf{w}_0, \mathbf{w}_M, \ldots$ are $w(=128)$-bit integers (= horizontal vectors in $\mathbb{F}_2^{128}$), and A, B, C, D are sparse 128×128 matrices over $\mathbb{F}_2$ for which $\mathbf{w}A, \mathbf{w}B, \mathbf{w}C, \mathbf{w}D$ can be computed by a few SIMD bit-operations. The choice of the suffixes $N-1$, $N-2$ is for speed: in the implementation of g, `W[0]` and `W[M]` are read from the array `W`, while the copies of `W[N-2]` and `W[N-1]` are kept in two 128-bit registers in the CPU, say `r1` and `r2`. Concretely speaking, we assign `r2` $\leftarrow$ `r1` and `r1` $\leftarrow$ "the result of (2)" at every generation, then `r2` (`r1`) keeps a copy of `W[N-2]` (`W[N-1]`, respectively). The merit of doing this is to use the pipeline effectively. To fetch `W[0]` and `W[M]` from memory takes some time. In the meantime, the CPU can compute $\mathbf{w}_{N-2}C$ and $\mathbf{w}_{N-1}D$, because copies of $\mathbf{w}_{N-2}$ and $\mathbf{w}_{N-1}$ are kept in the registers.

By trial and error, we searched for a set of parameters of SFMT, with the period being a multiple of $2^{19937}-1$ and having good equidistribution properties. The degree of recursion N is $\lceil 19937/128 \rceil = 156$, and the linear transformations A, B, C, D are as follows.

- $\mathbf{w}A := (\mathbf{w} \overset{128}{<<} 8) + \mathbf{w}$.

  This notation means that $\mathbf{w}$ is regarded as a single 128-bit integer, and $\mathbf{w}A$ is the result of the left-shift of $\mathbf{w}$ by 8 bits, ex-ored with $\mathbf{w}$ itself. The shift of a 128-bit integer and exor of 128-bit integers are both in Pentium SSE2 and PowerPC AltiVec SIMD instruction sets (SSE2 permits only a multiple of 8 as the amount of shifting). Note that the notation + means the exclusive-or in this article.

- $\mathbf{w}B := (\mathbf{w} \overset{32}{>>} 11)\&$(`BFFFFFF6 BFFAFFFF DDFECB7F DFFFFFEF`).

  This notation means that $\mathbf{w}$ is considered to be a quadruple of 32-bit integers, and each 32-bit integer is shifted to the right by 11 bits, (thus

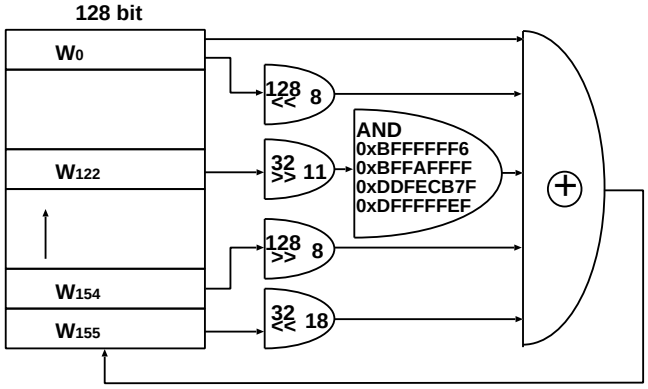

Figure 1: A circuit-like description of SFMT19937.

the eleven most significant bits are filled with 0s, for each 32-bit integer). The C-like notation & means the bitwise AND with a constant 128-bit integer, denoted in the hexadecimal form.

In the search, this constant is randomly generated as follows. Each bit in the 128-bit integer is independently randomly chosen, with the probability to choose 1 being 7/8. This is because we prefer to have more 1's for a denser feedback.

- $\mathbf{w}C := (\mathbf{w} \overset{128}{>>} 8)$.

  The notation $(\mathbf{w} \overset{128}{>>} 8)$ is the right shift of an 128-bit integer by 8 bits, similar to the first.

- $\mathbf{w}D := (\mathbf{w} \overset{32}{<<} 18)$.

  Similar to the second, $\mathbf{w}$ is cut into four pieces of 32-bit integers, and each of these is shifted by 18 bits to the left.

All these instructions are available in both Intel Pentium's SSE2 and PowerPC's AltiVec SIMD instruction sets. Figure 1 shows a concrete description of SFMT19937 generator with period a multiple of $2^{19937} - 1$.

3 How to select the recursion and parameters.

We wrote a code to compute the period and the dimensions of equidistribution (DE, see §3.2). Then, we search for a recursion with good DE admitting a fast implementation.

3.1 Computation of a Period

LFSR by the recursion (1) may be considered as an automaton, with a state space $S = (\mathbb{F}_2^w)^N$ and a state transition function $f : S \to S$ given by $(\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}) \mapsto (\mathbf{w}_1, \ldots, \mathbf{w}_{N-1}, g(\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}))$. As a w-bit integer generator, its output function is $o : S \to \mathbb{F}_2^w,\ (\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}) \mapsto \mathbf{w}_0$.

Let χ_f be the characteristic polynomial of $f : S \to S$. If χ_f is primitive, then the period of the state transition takes the maximal value $2^{\dim(S)} - 1$ [7, §3.2.2]. However, to check the primitivity, we need the integer factorization of this number, which is often hard for $\dim(S) = nw > 10000$. On the other hand, the primarity test is much easier than the factorization, so many huge primes of the form $2^p - 1$ have been found. Such a prime is called a Mersenne prime, and p is called the Mersenne exponent, which itself is a prime.

MT and WELL[14] discard some fixed r-bits from the array S, so that $nw-r$ is a Mersenne exponent. Then, the primitivity of χ_f is easily checked by the algorithm in [7, §3.2.2], avoiding the integer factorization.

SFMT adopted another method to avoid the integer factorization, the reducible transition method (RTM), which uses a reducible characteristic polynomial. This idea appeared in [5] [1][2], and applications in the present context are discussed in detail in another article [15], therefore we only briefly recall it.

Let p be the Mersenne exponent, and $N := \lceil p/w \rceil$. Then, we randomly choose parameters for the recursion of LFSR (1). By applying the Berlekamp-Massey Algorithm to the output sequence, we obtain the minimal polynomial of the transition function f. (Note that a direct computation of $\det(tI - f)$ is time-consuming because $\dim(S) = 19968$.)

By using a sieve, we remove all factors of small degree from χ_f, until we know that it has no irreducible factor of degree p, or that it has a (possibly reducible) factor of degree p. In the latter case, the factor is passed to the primitivity test described in [7, §3.2.2].

Suppose that we found a recursion with an irreducible factor of desired degree p in $\chi_f(t)$. Then, we have a factorization

$$\chi_f = \phi_p \phi_r,$$

where ϕ_p is a primitive polynomial of degree p and ϕ_r is a polynomial of degree $r \leq wN - p$. These are coprime, since we assume $p > r$. By putting $V_p := \mathrm{Ker}\phi_p(f)$ and $V_r := \mathrm{Ker}\phi_r(f)$, we have a decomposition into f-invariant subspaces

$$S = V_p \oplus V_r,$$

For any element $s \in S$, we denote $s = s_p + s_r$ for the corresponding decomposition.

The period length of the state transition is the least common multiple of that started from s_p and that started from s_r. Hence, if $s_p \neq 0$, then the period is a nonzero multiple of $2^p - 1$.

Thus, what we want to do is to avoid $s \in S$ with $s_p = 0$. This can be done for example by obtaining a basis of V_r, and to discard s if s lies in the span

of V_r. However, this consumes some CPU time and some memory, since each vector in V_r is 19937-dimensional.

A more efficient method is to use a simple sufficient condition on $s = (\mathbf{w}_0, \ldots, \mathbf{w}_{N-1})$ for $s_p \neq 0$.

Let

$$\begin{array}{rccc} g : S & \twoheadrightarrow & \mathbb{F}_2^{128} \\ \mathbf{s} & \longmapsto & \mathbf{w}_0 \end{array}$$

be the projection to the first 128-bit component, and consider $g(S_r) \subset \mathbb{F}_2^{128}$. Since $\dim(S_r) < 128$, $g(S_r) \subset \mathbb{F}_2^{128}$ is a proper subspace. Thus, there is a nonzero vector $\mathbf{p} \in \mathbb{F}_2^{128}$ that is orthogonal to $g(S_r)$ with respect to the standard inner product.

We call such $\mathbf{p}$ a *period certification vector*, which is used in the initialization as follows. After a random choice of $s = (\mathbf{w}_0, \ldots, \mathbf{w}_{N-1})$ at the initialization, we compute the inner product of $\mathbf{w}_0$ and $\mathbf{p}$. If it is 1, then $s_p \neq 0$ and we use s as the initial state. If it is 0, then we invert one bit in $\mathbf{w}_0$ so that the inner product becomes 1.

An algorithm to obtain a period certification vector is as follows. The image of a basis of S by the mapping $\phi_p(f)$ gives a generating set of V_r. By extracting linearly independent vectors, we have a basis of V_r. Then, a basis of the orthogonal space of V_r is obtained by a standard Gaussian elimination.

By computation, we obtained the following value.

Proposition 3.1 A 128-bit integer $\mathbf{p}$ =`13c9e684 00000000 00000000 00000001` (in the hexadecimal form) is a period certificate vector for SFMT19937.

This implies that if the initial state $s = (\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{N-1})$ satisfies $\mathbf{w}_0 \cdot \mathbf{p} = 1$, then the period of the SFMT19937 is a nonzero multiple of $2^{19937} - 1$.

Remark 3.2 The number of non-zero terms in $\chi_f(t)$ is an index measuring the amount of bit-mixing. In the case of SFMT19937, the number of nonzero terms is 6711, which is much larger than 135 of MT, but smaller than 8585 of WELL19937c [14].

3.2 Computation of the dimension of equidistribution

We briefly recall the definition of dimension of equidistribution (cf. [3]).

Definition 3.3 A periodic sequence with period P

$$\chi := \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{P-1}, \mathbf{x}_P = \mathbf{x}_0, \ldots$$

of v-bit integers is said to be *k-dimensionally equidistributed* if any kv-bit pattern occurs equally often as a k-tuple

$$(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+k-1})$$

for a period $i = 0, \ldots, P-1$. We allow an exception for the all-zero pattern, which may occur once less often. (This last loosening of the condition is technically necessary, because the zero state does not occur in an $\mathbb{F}_2$-linear generator). The largest value of such k is called the dimension of equidistribution (DE).

We want to generalise this definition slightly. We define the k-window set of the periodic sequence χ as

$$W_k(\chi) := \{(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+k-1}) \,|\, i = 0, 1, \ldots, P-1\},$$

which is considered as a *multi-set*, namely, the multiplicity of each element is considered.

For a positive integer m and a multi-set T, let us denote by $m \cdot T$ the multi-set where the multiplicity of each element in T is multiplied by m. Then, the above definition of equidistribution is equivalent to

$$W_k(\chi) = (m \cdot \mathbb{F}_2^{vk}) \setminus \{\mathbf{0}\},$$

where m is the multiplicity of the occurrences, and the operator $\setminus$ means that the multiplicity of $\mathbf{0}$ is subtracted by one.

Definition 3.4 In the above setting, if there exist a positive integer m and a multi-subset $D \subset (m \cdot \mathbb{F}_2^{vk})$ such that

$$W_k(\chi) = (m \cdot \mathbb{F}_2^{vk}) \setminus D,$$

we say that χ is k-dimensionally equidistributed with defect ratio $\#(D)/\#(m \cdot \mathbb{F}_2^{vk})$, where the cardinality is counted with multiplicity.

Thus, in Definition 3.3, the defect ratio up to $1/(P+1)$ is allowed to claim the dimension of equidistribution. If $P = 2^{19937} - 1$, then $1/(P+1) = 2^{-19937}$. In the following, the dimension of equidistribution allows the defect ratio up to 2^{-19937}.

For a w-bit integer sequence, its *dimension of equidistribution at v-bit accuracy* $k(v)$ is defined as the DE of the v-bit sequence, obtained by extracting the v MSBs from each of the w-bit integers. If the defect ratio is $1/(P+1)$, then there is an upper bound

$$k(v) \le \lfloor \log_2(P+1)/v \rfloor.$$

The gap between the realized $k(v)$ and the upper bound is called the dimension defect at v of the sequence, and denoted by

$$d(v) := \lfloor \log_2(P+1)/v \rfloor - k(v).$$

The summation of all the dimension defects at $1 \le v \le 32$ is called the total dimension defect, denoted by Δ.

There is a difficulty in computing $k(v)$ when a 128-bit integer generator is used as a 32-bit (or 64-bit) integer generator. SFMT generates a sequence

$\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ of 128-bit integers. Then, they are converted to a sequence of 32-bit integers $\mathbf{x}_0[0], \mathbf{x}_0[1], \mathbf{x}_0[2], \mathbf{x}_0[3], \mathbf{x}_1[0], \mathbf{x}_1[1], \ldots$, where $\mathbf{x}[0]$ is the 32 LSBs of $\mathbf{x}$, $\mathbf{x}[1]$ is the 33rd–64th bits, $\mathbf{x}[2]$ is the 65rd–96th bits, and $\mathbf{x}[3]$ is the 32 MSBs. This is the so-called little-endian system (see §8 for an implementation in a big-endian system).

Then, we need to modify the model automaton as follows. The state space is $S' := S \times \{0, 1, 2, 3\}$, the state transition function $f' : S' \to S'$ is

$$f'(s, i) := \begin{cases} (s, i+1) & (\text{ if } i < 3), \\ (f(s), 0) & (\text{ if } i = 3) \end{cases}$$

and the output function is

$$o' : S' \to \mathbb{F}_2^{32},\ ((\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}), i) \mapsto \mathbf{w}_0[i].$$

We fix $1 \leq v \leq w$, and let $o_k(s, i)$ be the k-tuple of the v MSBs of the consecutive k-outputs from the state (s, i).

Proposition 3.5 Assume that f is bijective. Let $k' = k'(v)$ denote the maximum k such that

$$o_k(-, i) : V_p \subset S \to \mathbb{F}_2^{kv}, \quad s \mapsto o_k(s, i) \tag{3}$$

are surjective for all $i = 0, 1, 2, 3$. Take the initial state s satisfying $s_p \neq 0$. Then, the 32-bit output sequence is at least $k'(v)$-dimensionally equidistributed with v-bit accuracy with defect ratio 2^{-p}.

Moreover, if $4 < k'(v) + 1$, then for any initial state with $s = s_p \neq 0$ (hence $s_r = 0$), the dimension of equidistribution with defect ratio 2^{-p} is exactly $k'(v)$.

Proof Take $s \in S$ with $s_p \neq 0$. Then, the orbit of s by f has the form of $(V_p - \{0\}) \times U \subset V_p \times V_r$, since $p > r$ and $2^p - 1$ is a prime. The surjectivity of the linear mapping $o_{k'}(-, i)$ implies that the image of

$$o_{k'}(-, i) : V_p \times U \to \mathbb{F}_2^{kv}$$

is $m \cdot \mathbb{F}_2^{kv}$ as a multi-set for some m. The defect comes from $0 \in V_p$, whose ratio in V_p is 2^{-p}. Then the first statement follows, since $W_{k'}(\chi)$ is the union of the images $o_{k'}(-, i)((V_p - \{0\}) \times U)$ for $i = 0, 1, 2, 3$.

For the latter half, we define L_i as the multiset of the image of $o_{k'+1}(-, i) : V_p \to \mathbb{F}_2^{(k'+1)v}$. Because of $s_r = 0$, we have $U = \{0\}$, and the union of $(L_i - \{0\})$ $(i = 0, 1, 2, 3)$ as a multi-set is $W_{k'+1}(\chi)$. If the sequence is $(k'+1)$-dimensionally equidistributed, then the multiplicity of each element in $W_{k'+1}(\chi)$ is at most $2^p \times 4/2^{(k'+1)v}$.

On the other hand, the multiplicity of an element in L_i is equal to the cardinality of the kernel of $o_{k'+1}(-, i)$. Let d_i be its dimension. Then by the dimension theorem, we have $d_i \geq p - (k' + 1)v$, and the equality holds if and only if $o_{k'+1}(-, i)$ is surjective. Thus, if there is a nonzero element $x \in \cap_{i=0}^{3} L_i$, then its multiplicity in $W_{k'+1}(\chi)$ is no less than $4 \times 2^{p-(k'+1)v}$, and since one of

$o_{k'+1}(-,i)$ is not surjective by the definition of k', its multiplicity actually exceeds $4\times 2^{p-(k'+1)v}$, which implies that the sequence is not $(k'+1)$-dimensionally equidistributed, and the proposition follows. Since the codimension of L_i is at most v, that of $\cap_{i=0}^{3} L_i$ is at most $4v$. The assumed inequality on k' implies the existence of nonzero element in the intersection. □

The dimension of equidistribution $k(v)$ depends on the choice of the initial state s. The above proposition implies that $k'(v)$ coincides with $k(v)$ for the worst choice of s under the condition $s_p \neq 0$. Thus, we adopt the following definition.

Definition 3.6 Let k be the maximum such that (3) is satisfied. We call this the dimension of equidistribution of v-bit accuracy, and denote it simply by $k(v)$. We have an upper bound $k(v) \leq \lfloor p/v \rfloor$.

We define the dimension defect at v by

$$d(v) := \lfloor p/v \rfloor - k(v) \text{ and } \Delta := \sum_{v=1}^{w} d(v).$$

We may compute $k'(v)$ by standard linear algebra. We used a more efficient algorithm based on a weighted norm, generalising [3].

Here we briefly recall the method in [3]. Let us denote the output v-bit sequence from an initial state s_0 by

$$\begin{aligned} & b_{ij} \in \mathbb{F}_2 \\ o(s_0) &= (b_{10}, b_{20}, ..., b_{v0}), \\ o(s_1) &= (b_{11}, b_{21}, ..., b_{v1}), \\ &\vdots \end{aligned}$$

We assign to the initial state $s_o \in S$ a v-dimensional vector with components in the formal power series $A = \mathbb{F}_2[[t]]$:

$$w(s_0) = \left(\sum_{j=0}^{\infty} b_{1j} t^j, \sum_{j=0}^{\infty} b_{2j} t^j, ..., \sum_{j=0}^{\infty} b_{vj} t^j, \right).$$

This assignment $w : S \to F^v$ is an $\mathbb{F}_2$-linear function.

We consider the formal Laurent series field $F = \mathbb{F}_2((t)) \supset A$, and define its norm and the norm on F^v by

$$\begin{aligned} \left| \sum_{i=-m}^{\infty} a_i t^i \right| &:= 2^m \quad (a_{-m} \neq 0), \quad |0| = 0 \\ ||(x_1, ...x_v)|| &:= \max_{i=1,2,...,v} \{|x_i|\}. \end{aligned}$$

The polynomial ring $\mathbb{F}_2[t^{-1}]$ is discrete in F, and consider an $\mathbb{F}_2[t^{-1}]$-lattice $L \subset F^v$ defined by

$$\begin{aligned} e_i &:= (0, ..., 0, t^{-1}, 0, ..., 0) \quad \text{the } i\text{-th component is } t^{-1}, \text{ others being } 0 \\ L &:= \mathbb{F}_2[t^{-1}]\langle e_1, ..., e_v, w(s)\rangle \end{aligned}$$

Basic theorems used in [3] are the following.

Theorem 3.7 Suppose that the PRNG satisfies the maximal period condition. If the covering radius of L is 2^{-k-1}, then the dimension of the equidistribution of the output v-bit sequence is k.

Theorem 3.8 The covering radius of L is 2^{-k-1} if and only if the shortest basis of L has the norm 2^{-k}.

We may apply this method to SFMT19937, but only as a 128-bit integer generator, since SFMT19937 is not an $\mathbb{F}_2$-linear generator as 32 or 64-bit integer generator. Instead, we need to check the surjectivity of

$$o_k(-, i) : V_p \to \mathbb{F}_2^{kv}$$

for $i = 0, 1, 2, 3$. For simplicity we treat only the case $i = 0$, since other cases follow similarly.

Let x_j be the j-th output 128-bit integer of SFMT19937, and let $b_{j,m} \in \mathbb{F}_2$ be its m-th bit (LSB considered 0-th bit). Then, the consecutive k tuples of most significant v bits of the 32-bit integer output sequence is the rectangular part in:

$$k \text{ tuples} \left\{ \begin{array}{rcl} & & \overbrace{\phantom{b_{j,31}, b_{j,30}, ..., b_{j,31-v+1}}}^{v \text{ bits}} \\ x_j[0] & : & \overbrace{b_{j,31}, b_{j,30}, ..., b_{j,31-v+1}}, b_{j,31-v}, ..., b_{j,0} \\ x_j[1] & : & b_{j,63}, b_{j,62}, ..., b_{j,63-v+1}, b_{j,63-v}, ..., b_{j,32} \\ & & \vdots \\ x_j[3] & : & b_{j,127}, b_{j,126}, ..., b_{j,127-v+1}, b_{j,127-v}, ..., b_{j,96} \\ & & \vdots \\ x_{j+\lceil \frac{k}{4} \rceil}[k \bmod 4] & : & ... \end{array} \right.$$

The corresponding part in the 128-bit sequence is marked by the parentheses in the following:

$$\left\lceil \frac{k}{4} \right\rceil \text{ tuples} \left\{ \begin{array}{rcl} & & \overbrace{\phantom{b_{j,127}, ...}}^{v \text{ bits}} \quad \overbrace{\phantom{b_{j,95}, ...}}^{v \text{ bits}} \quad \overbrace{\phantom{b_{j,63}, ...}}^{v \text{ bits}} \quad \overbrace{\phantom{b_{j,31}, ...}}^{v \text{ bits}} \\ x_j & : & \overbrace{b_{j,127}, ...}, ..., \overbrace{b_{j,95}, ...}, ..., \overbrace{b_{j,63}, ...}, ..., \overbrace{b_{j,31}, ...}, ..., b_{j,0} \\ & & \vdots \\ & & \qquad\qquad\qquad\qquad \overbrace{\phantom{b_{j,63}, ...}}^{v \text{ bits}} \quad \overbrace{\phantom{b_{j,31}, ...}}^{v \text{ bits}} \\ x_{j+\lceil \frac{k}{4} \rceil - 1} & : & b_{j,127}, ..., b_{j,95}, ..., \underbrace{\overbrace{b_{j,63}, ...}, ..., \overbrace{b_{j,31}, ...}, ..., b_{j,0}}_{(k \bmod 4) \text{ 32-bit words}} \end{array} \right.$$

If we remove the last $v \times (k \bmod 4)$ bits, then the rest $\frac{k}{4}4v$ bits among the 32-bit integers is identical to the same number of bits in the 128-bit integer sequence

$$\left\lceil \frac{k}{4} \right\rceil \text{ tuples} \left\{ \begin{array}{lll} x_j & : & \overbrace{b_{j,31},...}^{v \text{ bits}}, \overbrace{b_{j,63},...}^{v \text{ bits}}, \overbrace{b_{j,95},...}^{v \text{ bits}}, \overbrace{b_{j,127},...}^{v \text{ bits}}, \\ & & \vdots \end{array} \right. \tag{4}$$

We may apply the lattice method to this $4v$-bit sequence, and let k' be the dimension of the equidistribution. Then, we have an estimation of $k(v)$ as 32-bit integer sequence by

$$4k' \leq k(v) < 4(k'+1).$$

To obtain the exact value of $k(v)$, we introduce another norm, named *weighted norm*, on F^{4v}. We consider a $4v$-bit integer sequence as in (4), and define the norm of weight type u ($u = 0, 1, 2, 3$) on F^{4v} as follows.

$$||(x_1, ..., x_{4v})||_u := \max\{\max_{i=1}^{(4-u)v} \{|x_i|\}, \max_{i=4v-uv+1}^{4v} \{2|x_i|\}\}$$

If $u = 0$, then this is the supreme norm treated already. It is easy to check that this gives an ultra norm for any u.

Theorem 3.9 Let u to be an integer with $0 \leq u \leq 3$, and let F^v equipped with the weighted norm of type u. Then, the covering radius of L with respect to this norm is $\leq 2^{-r-1}$, if and only if

$$o(4r+u, 0) : V_p \to \mathbb{F}_2^{4r+u}$$

is surjective.

Proof The surjectivity is equivalent to that there are enough points in L so that its $(4r+u)v$ bits corresponding to the weight-type u assume every possible bit pattern. This implies that the covering radius of L is at most 2^{-r-1}.

The converse follows in the same way. □

As explained above, by applying the usual (non-weighted) lattice method to $4v$-bit sequence, we obtain a closest lower bound $4k' \leq k(v) < 4(k'+1)$. So, using the above theorem for $u = 1, 2, 3$, we can check whether $o(4k'+u, 0)$ is surjective or not, to obtain the maximum k_0 such that $o(k_0, 0)$ is surjective.

A slightly modified method gives the maximum k_i such that $o(k_i, i)$ is surjective (for each $i = 1, 2, 3$). Now, $k(v)$ is obtained as the minimum of k_i, $i = 0, 1, 2, 3$. Thus, the algorithm to compute $k(v)$ is as follows.

Input: v
Output: k
Loop i = 0, 1, 2, 3
  Loop weight-type u = 0, 1, 2, 3
    Compute the norm of the shortest basis of $4v$-bit integers

with respect to the weight-type u. Let 2^{-r-1} be the norm of the shortest basis. Put $k_{i,u} := 4r + u$.

End Loop

Let k_i be the maximum of $k_{i,u}$ $(u = 0, 1, 2, 3)$.

End Loop

Output the minimum value of k_i $(i = 0, 1, 2, 3)$ as k.

We use Lenstra's reduction method to obtain a shortest basis from a generating set. Since the lattice L is independent of the weight type u, in this algorithm, after obtaining a shortest basis with weight type 0, we may apply Lenstra's algorithm to this shortest basis with respect to the weight-type 1. This is much faster than starting from the generating set in the definition of L.

A similar algorithm is applicable when SFMT19937 is considered as a 64-bit integer sequence generator.

4 Comparison of speed

We compared two algorithms: MT19937 and SFMT19937, with implementations using and without using SIMD instructions.

We measured the speeds for four different CPUs: Pentium M 1.4GHz, Pentium IV 3GHz, AMD Athlon 64 3800+, and PowerPC G4 1.33GHz. In returning the random values, we used two different methods. One is sequential generation, where one 32-bit random integer is returned for one call. The other is block generation, where an array of random integers is generated for one call (cf. [7]). For detail, see §7 below.

We measured the consumed CPU time in second, for 10^8 generations of 32-bit integers. More precisely, in case of the block generation, we generate 10^5 of 32-bit random integers by one call, and it is iterated for 10^3 times. For sequential generation, the same 10^8 32-bit integers are generated, one per a call. We used the inline declaration `inline` to avoid the function call, and unsigned 32-bit, 64-bit integer types `uint32_t`, `uint64_t` defined in INTERNATIONAL STANDARD ISO/IEC 9899 : 1999(E) Programming Language-C, Second Edition (which we shall refer to as C99 in the rest of this article). Implementations without SIMD are written in C99, whereas those with SIMD use some standard SIMD extension of C99 supported by the compilers icl (Intel C compiler) and gcc.

Table 1 summarises the speed comparisons. The first four lines list the CPU time (in second) needed to generate 10^8 32-bit integers, for a Pentium-M CPU with the Intel C/C++ compiler. The first line lists the seconds for the block-generation scheme. The second line shows the ratio of CPU time to that of SFMT(SIMD). Thus, SFMT coded in SIMD is 2.10 times faster than MT coded in SIMD, and 3.77 times faster than MT without SIMD. The third line lists the seconds for the sequential generation scheme. The fourth line lists the ratio, with the basis taken at SFMT(SIMD) block-generation (not sequential). Thus,

CPU/compiler	return	MT	MT(SIMD)	SFMT	SFMT(SIMD)
Pentium-M	block	1.122	0.627	0.689	0.298
1.4GHz	(ratio)	3.77	2.10	2.31	1.00
Intel C/C++	seq	1.511	1.221	1.017	0.597
ver. 9.0	(ratio)	5.07	4.10	3.41	2.00
Pentium IV	block	0.633	0.391	0.412	0.217
3GHz	(ratio)	2.92	1.80	1.90	1.00
Intel C/C++	seq	1.014	0.757	0.736	0.412
ver. 9.0	(ratio)	4.67	3.49	3.39	1.90
Athlon 64 3800+	block	0.686	0.376	0.318	0.156
2.4GHz	(ratio)	4.40	2.41	2.04	1.00
gcc	seq	0.756	0.607	0.552	0.428
ver. 4.0.2	(ratio)	4.85	3.89	3.54	2.74
PowerPC G4	block	1.089	0.490	0.914	0.235
1.33GHz	(ratio)	4.63	2.09	3.89	1.00
gcc	seq	1.794	1.358	1.645	0.701
ver. 4.0.0	(ratio)	7.63	5.78	7.00	2.98

Table 1: The CPU time (sec.) for 10^8 generations of 32-bit integers, for four different CPUs and two different return-value methods. The ratio to the SFMT coded in SIMD is listed, too.

the block-generation of SFMT(SIMD) is 2.00 times faster than the sequential-generation of SFMT(SIMD).

Roughly speaking, in the block generation, SFMT(SIMD) is twice as fast as MT(SIMD), and four times faster than MT without using SIMD. Even in the sequential generation case, SFMT(SIMD) is still considerably faster than MT(SIMD).

Table 2 lists the CPU time for generating 10^8 32-bit integers, for four PRNGs from the GNU Scientific Library and two recent generators. They are re-coded with inline specification. Generators examined were: a multiple recursive generator `mrg` [8], linear congruential generators `rand48` and `rand`, a lagged fibonacci generator `random256g2`, a WELL generator `well` (WELL19937c in [14]), and a XORSHIFT generator `xor3` [13] [9]. The table shows that SFMT(SIMD) is faster than these PRNGs, except for the outdated linear congruential generator `rand`, the lagged-fibonacci generator `random256g2` (which is known to have poor randomness, cf. [12]), and `xor3` with a Pentium-M.

5 Comparison of Dimension Defects

Table 3 lists the dimension defects $d(v)$ of SFMT19937 (as a 32-bit integer generator) and of MT19937, for $v = 1, 2, \ldots, 32$. SFMT has smaller values of the defect $d(v)$ at 26 values of v. The converse holds for 6 values of v, but the

CPU	return	mrg	rand48	rand	random256g2	well	xor3
Pentium M	block	3.277	1.417	0.453	0.230	1.970	0.296
	seq	3.255	1.417	0.527	0.610	2.266	1.018
Pentium IV	block	2.295	1.285	0.416	0.121	0.919	0.328
	seq	2.395	1.304	0.413	0.392	1.033	0.702
Athlon	block	1.781	0.770	0.249	0.208	0.753	0.294
	seq	1.798	0.591	0.250	0.277	0.874	0.496
PowerPC	block	2.558	1.141	0.411	0.653	1.792	0.618
	seq	2.508	1.132	0.378	1.072	1.762	1.153

Table 2: The CPU time (sec.) for 10^8 generations of 32-bit integers, by six other PRNGs.

v	MT	SFMT	v	MT	SFMT	v	MT	SFMT	v	MT	SFMT
$d(1)$	0	0	$d(9)$	346	1	$d(17)$	549	543	$d(25)$	174	173
$d(2)$	0	*2	$d(10)$	124	0	$d(18)$	484	478	$d(26)$	143	142
$d(3)$	405	1	$d(11)$	564	0	$d(19)$	426	425	$d(27)$	115	114
$d(4)$	0	*2	$d(12)$	415	117	$d(20)$	373	372	$d(28)$	89	88
$d(5)$	249	2	$d(13)$	287	285	$d(21)$	326	325	$d(29)$	64	63
$d(6)$	207	0	$d(14)$	178	176	$d(22)$	283	282	$d(30)$	41	40
$d(7)$	355	1	$d(15)$	83	*85	$d(23)$	243	242	$d(31)$	20	19
$d(8)$	0	*1	$d(16)$	0	*2	$d(24)$	207	206	$d(32)$	0	*1

Table 3: Dimension defects $d(v)$ of MT19937 and SFMT19937 as a 32-bit integer generator. The mark * means that MT has a smaller defect than SFMT at that accuracy.

difference is small. The total dimension defect Δ of SFMT19937 as a 32-bit integer generator is 4188, which is smaller than the total dimension defect 6750 of MT19937.

We also computed the dimension defects of SFMT19937 as a 64-bit (128-bit) integer generator, and the total dimension defect Δ is 14089 (28676, respectively). In some applications, the distribution of LSBs is important. To check them, we inverted the order of the bits (i.e. the i-th bit is exchanged with the $(w-i)$-th bit) in each integer, and computed the total dimension defect. It is 10328 (21337, 34577, respectively) as a 32-bit (64-bit, 128-bit, respectively) integer generator. Throughout the experiments, $d'(v)$ is very small for $v \leq 11$. We consider that these values are satisfactorily small, since they are comparable with MT for which no statistical deviation related to the dimension defect has been reported, as far as we know.

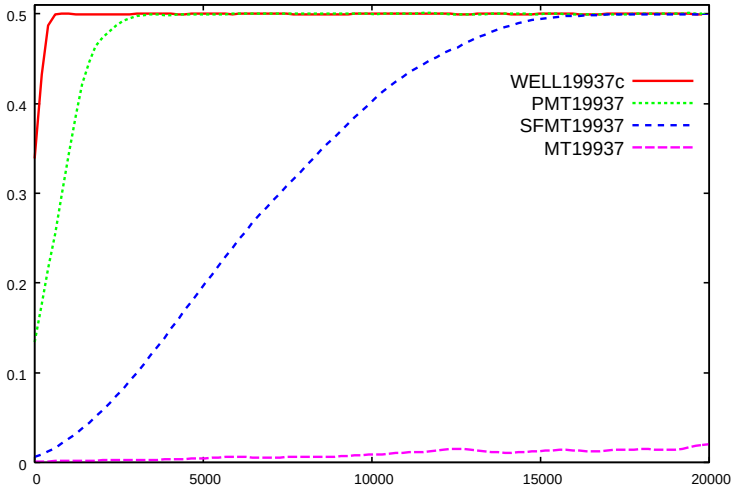

Figure 2: γ_k $(k = 0, \ldots, 20000)$: Starting from extreme 0-excess states, discard the first k outputs and then measure the ratio γ_k of 1's in the next 1000 outputs.

6 Recovery from 0-excess states

LFSR with a sparse feedback function g has the following phenomenon: if the bits in the state space contain too many 0's and few 1's (called a 0-excess state), then this tendency continues for considerable generations, since only a small part is changed in the state array at one generation, and the change is not well-reflected to the next generation because of the sparseness.

We measure the recovery time from 0-excess states, by the method introduced in [14], as follows.

1. Choose an initial state with only one bit being 1.
2. Generate k pseudorandom numbers, and discard them.
3. Compute the ratio of 1's among the next 32000 bits of outputs (i.e., in the next 1000 pseudorandom 32-bit integers).
4. Let γ_k be the average of the ratio over all such initial states.

We draw graphs of these ratio γ_k $(1 \le k \le 20000)$ in Figure 2 for the following generators: (1) WELL19937c, (2) PMT19937 [15], (3) SFMT19937, and (4) MT19937. Because of its dense feedback, WELL19937c shows the fastest recovery among the compared. SFMT is better than MT, since its recursion refers to the previously-computed words (i.e., `W[N-1]` and `W[N-2]`) that acquire new 1s, while MT refers only to the words generated long before (i.e., `W[M]` and `W[0]`). PMT19937 shows faster recovery than SFMT19937, since PMT19937 has two feedback loops (hence the name of Pulmonary Mersenne Twister).

The speed of recovery from 0-excess states is a trade-off with the speed of generation. Such 0-excess states will not happen practically, since the probability that 19937 random bits have less than 19937×0.4 of 1's is about 5.7×10^{-177}. The only plausible case is that a poor initialization scheme gives a 0-excess initial state. In a typical simulation, the number of initializations is far smaller than the number of generations, therefore we may spend more CPU time in the initialization than the generation. Once we avoid the 0-excess initial state by a well-designed initialization, then the recovery speed does not matter, in a practical sense. Consequently, the slower recovery of SFMT compared to WELL is not an issue, under the assumption that a good initialization scheme is provided.

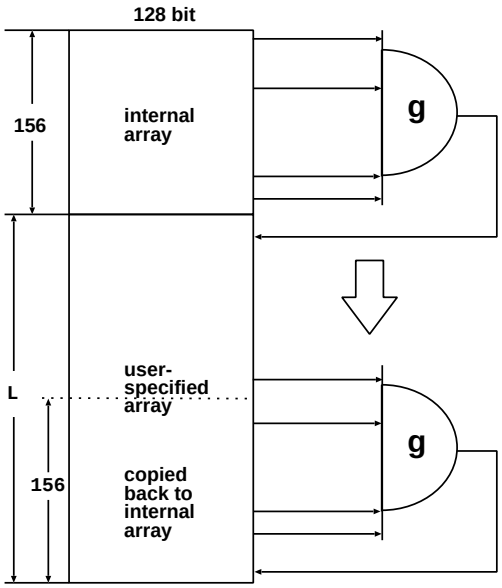

Figure 3: Block-generation scheme

7 Block-generation

Block-generation is introduced to avoid delay of function call. Moreover, branch prediction feature and multi-stage pipeline of Modern CPUs fits to its large counter `loop`, because `conditional branch` at the end of `loop` is assumed to be `jump back` by static branch prediction feature. When branch prediction hits, branch instruction doesn't break the pipeline.

A large scale simulation which consumes huge pseudorandom numbers can prepare them using block-generation before all or appropriate intervals.

In the block-generation scheme, a user of the PRNG specifies an array of w-bit integers of the length L, where $w = 32$ or 64 and L is specified by the

user. In the case of SFMT19937, wL should be a multiple of 128 and no less than $N \times 128$, since the array needs to accommodate the state space (note that $N = 156$). By calling the block generation function with the pointer to this array, w, and L, the routine fills up the array with pseudorandom integers, as follows. SFMT19937 keeps the state space S in an internal array of 128-bit integers of length 156. We concatenate this state array with the user-specified array, using the indexing technique. Then, the routine generates 128-bit integers in the user-specified array by recursion (2), as described in Figure 3, until it fills up the array. The last 156 128-bit integers are copied back to the internal array of SFMT19937. This makes the generation much faster than sequential generation (i.e., one generation per one call) as shown in Table 1.

8 Portability

Using CPU dependent features cause a portability problem. We prepare (1) a standard C code of SFMT, which uses functions specified in C99 only, (2) an optimized C code for Intel Pentium SSE2, and (3) an optimized C code for PowerPC AltiVec. The optimized codes require icl (Intel C Compiler) or gcc compiler with suitable options. Here we mention again that SFMT implemented in standard C code is faster than MT.

There is a problem of the endian when 128-bit integers are converted into 32-bit integers. When a 128-bit integer is stored as an array of 32-bit integers with length 4, in a little endian system such as Pentium, the 32 LSBs of the 128-bit integer come first. On the other hand, in a big endian system such as PowerPC, the 32 MSBs come first. The explanation above is based on the former. To assure the exactly same outputs for both endian systems as 32-bit integer generators, in the SIMD implementation for PowerPC, the recursion (2) is considered as a recursion on quadruples of 32-bit integers, rather than 128-bit integers, so that the content of the state array coincides both for little and big endian systems, as an array of 32-bit integers (not as 128-bit integers). Then, shift operations on 128-bit integers in PowerPC differs from those of Pentium. Fortunately, PowerPC supports arbitrary permutations of 16 blocks of 8-bit integers in a 128-bit register, which emulates the Pentium's shift by a multiple of 8.

9 Concluding remarks

We proposed SFMT pseudorandom number generator, which is a very fast generator with satisfactorily high dimensional equidistribution property.

9.1 Trade-off between speed and quality

It is difficult to measure the generation speed of a PRNG in a fair way, since it depends heavily on the circumstance. The WELL [14] generators have the best possible dimensions of equidistribution (i.e. $\Delta = 0$) for various periods ($2^{1024}-1$

to $2^{19937} - 1$). If we use the function call to PRNG for each generation, then a large part of the CPU time is consumed for handling the function call, and in the experiments in [14] or [13], WELL is not much slower than MT. On the other hand, if we avoid the function call, WELL is much slower than MT as seen in Table 1 and Table 2.

Since $\Delta = 0$, WELL has a better quality than MT or SFMT in a theoretical sense. However, one may argue whether this difference is observable or not. In the case of an $\mathbb{F}_2$-linear generator, the dimension of equidistribution $k(v)$ of v-bit accuracy means that there is no constant linear relation among the kv bits, but there exists a linear relation among the $(k+1)v$ bits, where kv bits ($(k+1)v$ bits) are taken from all the consecutive k integers ($k+1$ integers, respectively) by extracting the v MSBs from each. However, the existence of a linear relation does not necessarily mean the existence of some observable bias. According to [11], it requires 10^{28} samples to detect an $\mathbb{F}_2$-linear relation with 15 (or more) terms among 521 bits, by a standard statistical test. If the number of bits is increased, the necessary sample size is increased rapidly. Thus, it seems that $k(v)$ of SFMT19937 is sufficiently large, far beyond the level of the observable bias. On the other hand, the speed of the generator is observable. Thus, SFMT focuses more on the speed, for applications that require fast generations.

Acknowledgments

The author is very grateful to Professor Makoto Matsumoto for his beneficial advice and continuous encouragement throughout the course of this work.

References

[1] R.P. Brent and P. Zimmermann. Random number generators with period divisible by a mersenne prime. In *Computational Science and its Applications - ICCSA 2003*, volume 2667, pages 1–10, 2003.

[2] R.P. Brent and P. Zimmermann. Algorithms for finding almost irreducible and almost primitive trinomials. *Fields Inst. Commun.*, 41:91–102, 2004.

[3] R. Couture, P. L'Ecuyer, and S. Tezuka. On the distribution of k-dimensional vectors for simple and combined tausworthe sequences. *Math. Comp.*, 60(202):749–761, 1993.

[4] Sam Fuller. Motorola's AltiVec Technology. `http://www.freescale.com/files/32bit/doc/fact_sheet/ALTIVECWP.pdf`.

[5] M. Fushimi. Random number generation with the recursion $x_t = x_{t-3p} \oplus x_{t-3q}$. *Journal of Computational and Applied Mathematics*, 31:105–118, 1990.

[6] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. `http://www.intel.com/design/processor/manuals/248966.pdf`.

[7] D. E. Knuth. *The Art of Computer Programming. Vol.2. Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[8] P. L'Ecuyer. A search for good multiple recursive random number genarators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, April 1993.

[9] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[10] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.

[11] M. Matsumoto and T. Nishimura. A nonempirical test on the weight of pseudorandom number generators. In *Monte Carlo and Quasi-Monte Carlo methods 2000*, pages 381–395. Springer-Verlag, 2002.

[12] M. Matsumoto and T. Nishimura. Sum-discrepancy test on pseudorandom number generators. *Mathematics and Computers in Simulation*, 62(3-61):431–442, 2003.

[13] F. Panneton and P. L'Ecuyer. On the Xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, 2005.

[14] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear reccurences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16, 2006.

[15] M. Saito, H. Haramoto, F. Panneton, T. Nishimura, and M. Matsumoto. Pulmonary LFSR: pseudorandom number generators with multiple feedbacks and reducible transitions. 2006. submitted.

[16] SIMD from wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/SIMD`.

Appendix

To compile the following C program with main function for test, type

```
cc -O3 -DNORMAL -DMAIN sfmt19937.c
```

and to make **.o** file which will be linked with your main program, type

```
cc -O3 -DNORMAL -c sfmt19937.c
```

If you have gcc version 3.4 or later and your CPU supports SSE2,

```
gcc -O3 -msse2 -DSSE2 -DMAIN sfmt19937.c
```

will make a test program with SSE2 feature. If you are using Macintosh computer with PowerPC G4 or G5, and your gcc version is later 3.3 then

```
gcc -O3 -faltivec -DALTIVEC -DMAIN sfmt19937.c
```

will make a test program with AltiVec feature.

For Intel C Compiler for windows, the following command will make a test program.

```
icl /O3 /QxBN /DSSE2 /DMAIN sfmt19937.c
```

Listing 1: sfmt19937.h

```
#include <inttypes.h>

inline uint32_t gen_rand32(void);
inline uint64_t gen_rand64(void);
inline void fill_array32(uint32_t array[], int size);
inline void fill_array64(uint64_t array[], int size);
void init_gen_rand(uint32_t seed);
void init_by_array(uint32_t init_key[], int key_length);
```

Listing 2: sfmt19937.c

```
#include <string.h>
#include <stdio.h>
#include <assert.h>
#include "sfmt19937.h"

#define MEXP 19937  /* Mersenne Exponent */
#define WORDSIZE 128  /* word size */
#define N (MEXP / WORDSIZE + 1) /* internal array size (128 bit)*/
#define N32 (N * 4)  /* array size as 32 bit */
#define N64 (N * 2)  /* array size as 64 bit */

#define POS1 122  /* the pick up position */
#define SL1 18   /* shift left as 32-bit registers */
#define SL2 1   /* shift left as 128-bit registers */
#define SR1 11   /* shift right as 32-bit registers */
#define SR2 1   /* shift right as 128-bit registers */
#define MSK1 0xdfffffefU /* bit mask 1 */
#define MSK2 0xddfecb7fU /* bit mask 2 */
#define MSK3 0xbffaffffU /* bit mask 3 */
#define MSK4 0xbffffff6U /* bit mask 4 */
#define PCV1 0x00000001U /* period certification vector 1 */
#define PCV2 0x00000000U /* period certification vector 2 */
#define PCV3 0x00000000U /* period certification vector 3 */
#define PCV4 0x13c9e684U /* period certification vector 4 */

#ifdef NORMAL
static uint32_t sfmt[N][4]; /* 128-bit internal state array */
static uint32_t *psfmt32 = &sfmt[0][0];/* 32bit pointer */
static uint64_t *psfmt64 = (uint64_t *)&sfmt[0][0];/* 64bit pointer */
#endif /* NORMAL */
#ifdef SSE2
#include <emmintrin.h>
static __m128i sfmt[N];
static uint32_t *psfmt32 = (uint32_t *)&sfmt[0];/* 32bit pointer */
static uint64_t *psfmt64 = (uint64_t *)&sfmt[0];/* 64bit pointer */
#endif /* SSE2 */
#ifdef ALTIVEC
static vector unsigned int sfmt[N];
static uint32_t *psfmt32 = (uint32_t *)&sfmt[0];/* 32bit pointer */
static uint64_t *psfmt64 = (uint64_t *)&sfmt[0];/* 64bit pointer */
#endif /* ALTIVEC */
static int idx;   /* index counter (32-bit) */
static int initialized = 0; /* initialized flag */
```

```
44 static int big_endian;  /* endian flag */
45 static uint32_t pcv[4] = {PCV1, PCV2, PCV3, PCV4};
46
47 #ifdef NORMAL
48 struct W128_T {   /* 128-bit data structure */
49     uint32_t a[4];
50 };
51 typedef struct W128_T w128_t; /* 128-bit data type */
52
53 inline static void rshift128(uint32_t out[4], const uint32_t in[4],
54                              int shift);
55 inline static void lshift128(uint32_t out[4], const uint32_t in[4],
56                              int shift);
57 inline static void gen_rand_array(w128_t array[], int size);
58 #endif /* NORMAL */
59 #ifdef SSE2
60 inline static __m128i mm_recursion(__m128i *a, __m128i *b,
61                                    __m128i c, __m128i d, __m128i mask);
62 inline static void gen_rand_array(__m128i array[], int size);
63 #endif /* SSE2 */
64 #ifdef ALTIVEC
65 inline static void gen_rand_array(vector unsigned int array[], int size);
66 inline static vector unsigned int vec_recursion(vector unsigned int a,
67                                                 vector unsigned int b,
68                                                 vector unsigned int c,
69                                                 vector unsigned int d);
70 inline static void vec_swap(vector unsigned int array[], uint32_t size);
71 #endif /* ALTIVEC */
72 inline static void gen_rand_all(void);
73 static void endian_check(void);
74 static void period_certification(void);
75
76 #ifdef NORMAL
77 /**
78  * This function simulates SIMD 128-bit right shift by the standard C.
79  * The 128-bit integer given in in[4] is shifted by (shift * 8) bits.
80  * This function simulates the LITTLE ENDIAN SIMD.
81  */
82 inline static void rshift128(uint32_t out[4], const uint32_t in[4],
83                              int shift) {
84     uint64_t th, tl, oh, ol;
85
86     th = ((uint64_t)in[3] << 32) | ((uint64_t)in[2]);
87     tl = ((uint64_t)in[1] << 32) | ((uint64_t)in[0]);
88
89     oh = th >> (shift * 8);
90     ol = tl >> (shift * 8);
91     ol |= th << (64 - shift * 8);
92     out[1] = (uint32_t)(ol >> 32);
93     out[0] = (uint32_t)ol;
94     out[3] = (uint32_t)(oh >> 32);
95     out[2] = (uint32_t)oh;
96 }
97
98 /**
99  * This function simulates SIMD 128-bit left shift by the standard C.
100  * The 128-bit integer given in in[4] is shifted by (shift * 8) bits.
101  * This function simulates the LITTLE ENDIAN SIMD.
102  */
103 inline static void lshift128(uint32_t out[4], const uint32_t in[4],
104                              int shift) {
105     uint64_t th, tl, oh, ol;
106
107     th = ((uint64_t)in[3] << 32) | ((uint64_t)in[2]);
108     tl = ((uint64_t)in[1] << 32) | ((uint64_t)in[0]);
109
110     oh = th << (shift * 8);
111     ol = tl << (shift * 8);
112     oh |= tl >> (64 - shift * 8);
113     out[1] = (uint32_t)(ol >> 32);
114     out[0] = (uint32_t)ol;
115     out[3] = (uint32_t)(oh >> 32);
116     out[2] = (uint32_t)oh;
117 }
118 #endif /* NORMAL */
119 /**
120  * This function represents the recursion formula.
121  */
122 #ifdef NORMAL
123 inline static void do_recursion(uint32_t r[4], uint32_t a[4], uint32_t b[4],
124                                 uint32_t c[4], uint32_t d[4]) {
125     uint32_t x[4];
126     uint32_t y[4];
127
128     lshift128(x, a, SL2);
129     rshift128(y, c, SR2);
130     r[0] = a[0] ^ x[0] ^ ((b[0] >> SR1) & MSK1) ^ y[0] ^ (d[0] << SL1);
131     r[1] = a[1] ^ x[1] ^ ((b[1] >> SR1) & MSK2) ^ y[1] ^ (d[1] << SL1);
132     r[2] = a[2] ^ x[2] ^ ((b[2] >> SR1) & MSK3) ^ y[2] ^ (d[2] << SL1);
```

```
    r[3] = a[3] ^ x[3] ^ ((b[3] >> SR1) & MSK4) ^ y[3] ^ (d[3] << SL1);
}
#endif /* NORMAL */
#ifdef SSE2
inline static
#if defined(__GNUC__)
__attribute__((always_inline))
#endif
    __m128i mm_recursion(__m128i *a, __m128i *b,
                         __m128i c, __m128i d, __m128i mask) {
    __m128i v, x, y, z;

    x = _mm_load_si128(a);
    y = _mm_srli_epi32(*b, SR1);
    z = _mm_srli_si128(c, SR2);
    v = _mm_slli_epi32(d, SL1);
    z = _mm_xor_si128(z, x);
    z = _mm_xor_si128(z, v);
    x = _mm_slli_si128(x, SL2);
    y = _mm_and_si128(y, mask);
    z = _mm_xor_si128(z, x);
    z = _mm_xor_si128(z, y);
    return z;
}
#endif /* SSE2 */
#ifdef ALTIVEC
inline static __attribute__((always_inline))
    vector unsigned int vec_recursion(vector unsigned int a,
                                      vector unsigned int b,
                                      vector unsigned int c,
                                      vector unsigned int d) {

    const vector unsigned int sl1
        = (vector unsigned int)(SL1, SL1, SL1, SL1);
    const vector unsigned int sr1
        = (vector unsigned int)(SR1, SR1, SR1, SR1);
    const vector unsigned int mask = (vector unsigned int)
    (MSK1, MSK2, MSK3, MSK4);
    const vector unsigned char perm_sl = (vector unsigned char)
    (1, 2, 3, 23, 5, 6, 7, 0, 9, 10, 11, 4, 13, 14, 15, 8);
    const vector unsigned char perm_sr = (vector unsigned char)
    (7, 0, 1, 2, 11, 4, 5, 6, 15, 8, 9, 10, 17, 12, 13, 14);

    vector unsigned int v, w, x, y, z;
    x = vec_perm(a, perm_sl, perm_sl);
    v = a;
    y = vec_sr(b, sr1);
    z = vec_perm(c, perm_sr, perm_sr);
    w = vec_sl(d, sl1);
    z = vec_xor(z, w);
    y = vec_and(y, mask);
    v = vec_xor(v, x);
    z = vec_xor(z, y);
    z = vec_xor(z, v);
    return z;
}
#endif /* ALTIVEC */
/**
 * This function fills the internal state array with psedorandom
 * integers.
 */
#ifdef NORMAL
inline static void gen_rand_all(void) {
    int i;
    uint32_t *r1, *r2;

    r1 = sfmt[N - 2];
    r2 = sfmt[N - 1];
    for (i = 0; i < N - POS1; i++) {
        do_recursion(sfmt[i], sfmt[i], sfmt[i + POS1], r1, r2);
        r1 = r2;
        r2 = sfmt[i];
    }
    for (; i < N; i++) {
        do_recursion(sfmt[i], sfmt[i], sfmt[i + POS1 - N], r1, r2);
        r1 = r2;
        r2 = sfmt[i];
    }
}
#endif /* NORMAL */
#ifdef SSE2
inline void gen_rand_all(void) {
    int i;
    __m128i r, r1, r2, mask;
    mask = _mm_set_epi32(MSK4, MSK3, MSK2, MSK1);

    r1 = _mm_load_si128(&sfmt[N - 2]);
    r2 = _mm_load_si128(&sfmt[N - 1]);
    for (i = 0; i < N - POS1; i++) {
        r = mm_recursion(&sfmt[i], &sfmt[i + POS1], r1, r2, mask);
```

```
        _mm_store_si128(&sfmt[i], r);
        r1 = r2;
        r2 = r;
    }
    for (; i < N; i++) {
        r = mm_recursion(&sfmt[i], &sfmt[i + POS1 - N], r1, r2, mask);
        _mm_store_si128(&sfmt[i], r);
        r1 = r2;
        r2 = r;
    }
}
#endif /* SSE2 */
#ifdef ALTIVEC
inline static void gen_rand_all(void) {
    int i;
    vector unsigned int r, r1, r2;

    r1 = sfmt[N - 2];
    r2 = sfmt[N - 1];
    for (i = 0; i < N - POS1; i++) {
        r = vec_recursion(sfmt[i], sfmt[i + POS1], r1, r2);
        sfmt[i] = r;
        r1 = r2;
        r2 = r;
    }
    for (; i < N; i++) {
        r = vec_recursion(sfmt[i], sfmt[i + POS1 - N], r1, r2);
        sfmt[i] = r;
        r1 = r2;
        r2 = r;
    }
}
#endif /*ALTIVEC */
/**
 * This function fills the user-specified array with psedorandom
 * integers.
 * @param array an 128-bit array to be filled by pseudorandom numbers.
 * @param size number of 128-bit pesudorandom numbers to be generated.
 */
#ifdef NORMAL
inline static void gen_rand_array(w128_t array[], int size) {
    int i;
    uint32_t *r1, *r2;

    r1 = sfmt[N - 2];
    r2 = sfmt[N - 1];
    for (i = 0; i < N - POS1; i++) {
        do_recursion(array[i].a, sfmt[i], sfmt[i + POS1], r1, r2);
        r1 = r2;
        r2 = array[i].a;
    }
    for (; i < N; i++) {
        do_recursion(array[i].a, sfmt[i], array[i + POS1 - N].a, r1, r2);
        r1 = r2;
        r2 = array[i].a;
    }
    for (; i < size; i++) {
        do_recursion(array[i].a, array[i - N].a, array[i + POS1 - N].a, r1, r2);
        r1 = r2;
        r2 = array[i].a;
    }
}
#endif /* NORMAL */
#ifdef SSE2
inline static void gen_rand_array(__m128i array[], int size) {
    int i, j;
    __m128i r, r1, r2, mask;
    mask = _mm_set_epi32(MSK4, MSK3, MSK2, MSK1);

    r1 = _mm_load_si128(&sfmt[N - 2]);
    r2 = _mm_load_si128(&sfmt[N - 1]);
    for (i = 0; i < N - POS1; i++) {
        r = mm_recursion(&sfmt[i], &sfmt[i + POS1], r1, r2, mask);
        _mm_store_si128(&array[i], r);
        r1 = r2;
        r2 = r;
    }
    for (; i < N; i++) {
        r = mm_recursion(&sfmt[i], &array[i + POS1 - N],
                         r1, r2, mask);
        _mm_store_si128(&array[i], r);
        r1 = r2;
        r2 = r;
    }
    /* main loop */
    for (; i < size - N; i++) {
        r = mm_recursion(&array[i - N], &array[i + POS1 - N],
                         r1, r2, mask);
        _mm_store_si128(&array[i], r);
        r1 = r2;
```

```
313             r2 = r;
314         }
315         for (j = 0; j < 2 * N - size; j++) {
316             r = _mm_load_si128(&array[j + size - N]);
317             _mm_store_si128(&sfmt[j], r);
318         }
319         for (; i < size; i++) {
320             r = mm_recursion(&array[i - N], &array[i + POS1 - N],
321                              r1, r2, mask);
322             _mm_store_si128(&array[i], r);
323             _mm_store_si128(&sfmt[j++], r);
324             r1 = r2;
325             r2 = r;
326         }
327     }
328     #endif /* SSE2 */
329     #ifdef ALTIVEC
330     inline static void gen_rand_array(vector unsigned int array[], int size)
331     {
332         int i, j;
333         vector unsigned int r, r1, r2;
334
335         r1 = sfmt[N - 2];
336         r2 = sfmt[N - 1];
337         for (i = 0; i < N - POS1; i++) {
338             r = vec_recursion(sfmt[i], sfmt[i + POS1], r1, r2);
339             array[i] = r;
340             r1 = r2;
341             r2 = r;
342         }
343         for (; i < N; i++) {
344             r = vec_recursion(sfmt[i], array[i + POS1 - N], r1, r2);
345             array[i] = r;
346             r1 = r2;
347             r2 = r;
348         }
349         /* main loop */
350         for (; i < size - N; i++) {
351             r = vec_recursion(array[i - N], array[i + POS1 - N], r1, r2);
352             array[i] = r;
353             r1 = r2;
354             r2 = r;
355         }
356         for (j = 0; j < 2 * N - size; j++) {
357             sfmt[j] = array[j + size - N];
358         }
359         for (; i < size; i++) {
360             r = vec_recursion(array[i - N], array[i + POS1 - N], r1, r2);
361             array[i] = r;
362             sfmt[j++] = r;
363             r1 = r2;
364             r2 = r;
365         }
366     }
367     inline static void vec_swap(vector unsigned int array[], uint32_t size)
368     {
369         int i;
370         const vector unsigned char perm = (vector unsigned char)
371             (4, 5, 6, 7, 0, 1, 2, 3, 12, 13, 14, 15, 8, 9, 10, 11);
372
373         for (i = 0; i < size; i++) {
374             array[i] = vec_perm(array[i], perm, perm);
375         }
376     }
377     #endif /* ALTIVEC */
378     /**
379      * This function checks ENDIAN of CPU and set big_endian flag.
380      */
381     static void endian_check(void) {
382         uint32_t a[2] = {0, 1};
383         uint64_t *pa;
384
385         pa = (uint64_t *)a;
386         if (*pa == 1) {
387             big_endian = 1;
388         } else {
389             big_endian = 0;
390         }
391     }
392
393     /**
394      * This function generates and returns 32-bit pseudorandom number.
395      * init_gen_rand or init_by_array must be called before this function.
396      * @return 32-bit pseudorandom number
397      */
398     inline uint32_t gen_rand32(void)
399     {
400         uint32_t r;
401
402         assert(initialized);
```

```
    if (idx >= N32) {
        gen_rand_all();
        idx = 0;
    }
    r = psfmt32[idx++];
    return r;
}

/**
 * This function generates and returns 64-bit pseudorandom number.
 * init_gen_rand or init_by_array must be called before this function.
 * The function gen_rand64 should not be called after gen_rand32,
 * unless an initialization is again executed.
 * @return 64-bit pseudorandom number
 */
inline uint64_t gen_rand64(void)
{
    uint32_t r1, r2;

    assert(initialized);
    assert(idx % 2 == 0);

    if (idx >= N32) {
        gen_rand_all();
        idx = 0;
    }
    r1 = psfmt32[idx];
    r2 = psfmt32[idx + 1];
    idx += 2;
    return ((uint64_t)r2 << 32) | r1;
}

/**
 * This function generates pseudorandom 32-bit integers in the
 * specified array[] by one call. The number of pseudorandom integers
 * is specified by the argument size, which must be at least 624 and a
 * multiple of four. The generation by this function is much faster
 * than the following gen_rand function.
 *
 * @param array an array where pseudorandom 32-bit integers are filled
 * by this function. The pointer to the array must be \b "aligned"
 * (namely, must be a multiple of 16) in the SIMD version, since it
 * refers to the address of a 128-bit integer. In the standard C
 * version, the pointer is arbitrary.
 *
 * @param size the number of 32-bit pseudorandom integers to be
 * generated. size must be a multiple of 4, and greater than or equal
 * to 624.
 */
inline void fill_array32(uint32_t array[], int size)
{
    assert(initialized);
    assert(idx == N32);
    assert(size % 4 == 0);
    assert(size >= N32);
#ifdef NORMAL
    gen_rand_array((w128_t *)array, size / 4);
    memcpy(psfmt32, array + size - N32, sizeof(uint32_t) * N32);
#endif
#ifdef SSE2
    gen_rand_array((__m128i *)array, size / 4);
#endif
#ifdef ALTIVEC
    gen_rand_array((vector unsigned int *)array, size / 4);
#endif
    idx = N32;
}

/**
 * This function generates pseudorandom 64-bit integers in the
 * specified array[] by one call. The number of pseudorandom integers
 * is specified by the argument size, which must be at least 312 and a
 * multiple of two. The generation by this function is much faster
 * than the following gen_rand function.
 *
 * @param array an array where pseudorandom 64-bit integers are filled
 * by this function. The pointer to the array must be "aligned"
 * (namely, must be a multiple of 16) in the SIMD version, since it
 * refers to the address of a 128-bit integer. In the standard C
 * version, the pointer is arbitrary.
 *
 * @param size the number of 64-bit pseudorandom integers to be
 * generated. size must be a multiple of 2, and greater than or equal
 * to 312.
 */
inline void fill_array64(uint64_t array[], int size)
{
    assert(initialized);
    assert(idx == N32);
    assert(size % 2 == 0);
```

```
    assert(size >= N64);
#ifdef NORMAL
    gen_rand_array((w128_t *)array, size / 2);
    memcpy(psfmt64, array + size - N64, sizeof(uint64_t) * N64);
    if (big_endian) {
        int i;
        uint32_t x;
        uint32_t *pa;
        pa = (uint32_t *)array;
        for (i = 0; i < size * 2; i += 2) {
            x = pa[i];
            pa[i] = pa[i + 1];
            pa[i + 1] = x;
        }
    }
#endif
#ifdef SSE2
    gen_rand_array((__m128i *)array, size / 2);
#endif
#ifdef ALTIVEC
    gen_rand_array((vector unsigned int *)array, size / 2);
    vec_swap((vector unsigned int *)array, size / 2);
#endif
    idx = N32;
}

/**
 * This function initializes the internal state array with a 32-bit
 * integer seed.
 */
void init_gen_rand(uint32_t seed)
{
    int i;

    psfmt32[0] = seed;
    for (i = 1; i < N32; i++) {
        psfmt32[i] = 1812433253UL
            * (psfmt32[i - 1] ^ (psfmt32[i - 1] >> 30)) + i;
    }
    idx = N32;
    endian_check();
    period_certification();
    initialized = 1;
}

/**
 * This function certificate non-zero multiple of period 2^{19937}
 */
static void period_certification(void) {
    int inner = 0;
    int i, j;
    uint32_t work;

    for (i = 0; i < 4; i++) {
        work = psfmt32[i] & pcv[i];
        for (j = 0; j < 32; j++) {
            inner ^= work & 1;
            work = work >> 1;
        }
    }
    /* check OK */
    if (inner == 1) {
        return;
    }
    /* check NG, and modification */
    for (i = 0; i < 4; i++) {
        work = 1;
        for (j = 0; j < 32; j++) {
            if ((work & pcv[i]) != 0) {
                psfmt32[i] ^= work;
                return;
            }
            work = work << 1;
        }
    }
}

#ifdef MAIN
#define BLOCK_SIZE 100000
#ifdef NORMAL
static uint64_t array[BLOCK_SIZE / 2][2];
#endif
#ifdef SSE2
static __m128i array[BLOCK_SIZE / 4];
#endif
#ifdef ALTIVEC
static vector unsigned int array[BLOCK_SIZE / 4];
#endif
int main(void) {
    int i;
```

```
583     uint32_t *array32 = (uint32_t *)array;
584     uint64_t *array64 = (uint64_t *)array;
585     uint32_t r32;
586     uint64_t r64;
587 
588     /* 32 bit generation */
589     init_gen_rand(1234);
590     fill_array32(array32, BLOCK_SIZE);
591     init_gen_rand(1234);
592     for (i = 0; i < 1000; i++) {
593         printf("%10"PRIu32"␣", array32[i]);
594         if (i % 5 == 4) {
595             printf("\n");
596         }
597         r32 = gen_rand32();
598         if (r32 != array32[i]) {
599             printf("\nmismatch␣at␣%d␣array32:%"PRIx32
600                    "␣gen:%"PRIx32"\n", i, array32[i], r32);
601             return 1;
602         }
603     }
604     /* 64 bit generation */
605     init_gen_rand(1234);
606     fill_array64(array64, BLOCK_SIZE / 2);
607     init_gen_rand(1234);
608     for (i = 0; i < 1000; i++) {
609         printf("%20"PRIu64"␣", array64[i]);
610         if (i % 3 == 2) {
611             printf("\n");
612         }
613         r64 = gen_rand64();
614         if (r64 != array64[i]) {
615             printf("\nmismatch␣at␣%d␣array32:%"PRIx64
616                    "␣gen:%"PRIx64"\n", i, array64[i], r64);
617             return 1;
618         }
619     }
620     printf("\n");
621     return 0;
622 }
623 #endif
```