免乘法線性同餘亂數產生 *

Multiplication-Free Linear Congruential Random Number Generators

吳培基

國立澎湖海事管理專科學校資訊工程科

澎湖縣880馬公市六合路300號

Email: pcwu@npit.edu.tw

摘 要

電腦模擬已廣泛應用於科學及工程領域這些應用要求高速、高品質的亂數產生器。各式線性同餘亂數產生器是經廣泛使用及測試的產生器之一。本文提出採用少量 2 的冪次的項次為乘數的線性同餘亂數產生器這些產生器的製作不需乘法指令，因此有很高的可攜性及執行效率。本文報告了五個亂數產生器的統計及速度測試的結果我們的結果顯示免乘法線性同餘亂數產生器可達到高速且並未降低品質。這些亂數產生器的製作也可採用平行指令來作向量化。本文並提出一線性同餘亂數產生器的硬體設計。

關鍵詞：統計測試，可攜性，效率，平行指令，硬體設計。

Abstract

Computer simulation has been widely used in scientific and engineering applications. These applications demand efficient, high-quality random number generators. Linear congruential random number generators in various forms are one of the most widely used and extensively studied. This paper presents linear congruential generators with multipliers having a few of $\pm 2^k$ terms. These generators can be implemented with no multiplication. They are portable and very fast. The statistical and timing results of five example generators are presented. Our result shows that multiplication-free linear congruential generators can achieve high speed with no degradation on quality. Vectorization of these generators using parallel instructions is briefly addressed. The hardware design of a generator is also presented.

Keywords: statistical tests, portability, efficiency, parallel instruction, hardware design.

1. Introduction

Computer simulation has been widely used in scientific and engineering applications. These applications demand efficient, high-quality random number generators. Linear congruential random number generators in various forms are one of the most widely used and extensively studied. Their characteristics are well known, and their execution speeds are acceptable. They have also been included in most mathematical packages.

The basic form of linear congruential generators is as follows

$$X_i = a \cdot X_{i-1} + c \bmod m \quad \text{.....(1)}$$

Eq. (1) includes 1 multiplication, 1 addition, and 1 division. The parameters a, c, and m determine the quality, the efficiency, and the cycle length of the generator. There are three well-known forms [1]:

$X_i = a \cdot X_{i-1} + c \bmod 2^N$, c is odd, $a \equiv 1 \pmod 4$, cycle length 2^N .....(2)

$X_i = a \cdot X_{i-1} \bmod 2^N$, X_0 is odd, $a \equiv 3, 5 \pmod 8$, cycle length 2^{N-2} .....(3)

$X_i = a \cdot X_{i-1} \bmod p$, $X_0>0$, p is a prime, a is a primitive root of GF(p), cycle length $p - 1$ .....(4)

N is usually the length of computer words, e.g., 32, 48, 64. Eqs. (3) and (4) remove the addition in Eq. (1). Eqs. (2) and (3) use powers of 2 as moduli and remove the divisions. Eq. (4) typically uses a Mersenne prime $p = 2^s - 1$ and reduces the division to a shift and an add with carry [12]. Eq. (4) can also be extended to n-dimension [8, p.28]

$$X_i = (a_1 \cdot X_{i-1} + \ldots + a_n \cdot X_{i-n}) \bmod p. \quad \text{.....(5)}$$

The cycle length is $p^n - 1$. The initial condition is as follows $(X_0, \ldots, X_{n-1}) \neq (0, \ldots, 0)$; and $f(x) = x^n - a_1 x^{n-1} - \ldots - a_n$ is a primitive polynomial of GF(p^n). Using a trinomial $f(x)$, Eq. (5) contains only two terms [6, 9]:

$$X_i = (a_d \cdot X_{i-d} + a_n \cdot X_{i-n}) \bmod p \quad \text{.....(6)}$$

Eqs. (5) and (6) have very long cycle lengths.

One way to further speeding up linear congruential generators is by selecting multipliers that can be efficient implemented. Knuth [8, p. 22-24] discusses Eq. (2) with $a = 2^s + 1$ and c=1, and finds that this kind of generators are bad. Knuth [8, p.102, Table 1] applies the spectral test to Eq. (3) with multipliers such as $a = 2^{23} + 2^{12} + 5$, and finds that these generators perform very bad in 4-dimension. Knuth [8, p.170] finally suggests that multipliers without special forms would be used. Some research [9, 10, 14] proposes portable implementations for the double word product $(a \cdot X_{i-1})$ in Eq. (4), which is usually coded in assembly languages.

* This research was supported in part by National Science Council, Taiwan, R.O.C., under Contract No. NSC 88-2218-E-346-002.

Unfortunately, all these implementations slow down the speed.

Wu [15] proposes generators of Eq. (4) with multipliers $\pm 2^{k1} \pm 2^{k2}$. These generators can be implemented with no multiplication. They are portable and very fast. This paper extends the technique to other forms of linear congruential generators. The statistical and timing results of five example generators are presented. Our result shows that multiplication-free linear congruential generators can achieve high speed with no degradation on quality. Vectorization of these generators u sing parallel instructions [13] is addressed in brief. The hardware design of a generator is also presented.

2. The Design of Multiplication-Free Generators

We choose multipliers that contain a few o $\pm 2^k$ terms, e.g., multipliers $\pm 2^{k1} \pm 2^{k2} + 5$, $\pm 2^{k1} \pm 2^{k2} + 3$, $\pm 2^{k1} \pm 2^{k2} + 1$ for Eqs. (2) and (3), and $\pm 2^{k1} \pm 2^{k2}$ for Eqs. (4), (5), and (6). All these multipliers contain the terms $\pm 2^{k1} \pm 2^{k2}$. The advantages of these multipliers are as follows:

1. High speed. A few adds and shifts are usually faster than a multiplication in most machines. Since shifts can be hard-wired by swapping data lines, hardware implementations of these generators need only adders.
2. Numerous multipliers. For example, $m = 2^{32}$ or $m = 2^{31} - 1$, there are about three thousands of such multipliers. It is hopefully to find adequate ones from these multipliers.
3. High portability. The resulting generators mainly use shifts and adds/subtracts. All these operations can be programmed using a high-level language such as the C programming language [7].
4. High parallelism degree. Using multimedia extension instructions 13] such as parallel shifts, adds, and subtracts, we can generate two (or more) random numbers at one time. Let the parallelism degree be L. The Eq. (6) can be vectorized if $d \geq L$. The condition for Eqs. (2)-(4) is as follows $A \equiv a^L \pmod m$; A is of the form $\pm 2^{k1} \pm 2^{k2}$; and a passes the spectral test. The generators of Eqs. (2)-(4) can be vectorized using Anderson [1, p. 239, Method I]:

   Let $X(L; 1)$ be a vector $x_0, a\,x_0 \pmod m, a^2 x_0 \pmod m, \ldots, a^{L-1} x_0 \pmod m$].

   $X'(L; 1) = A \cdot X(L; 1) \pmod m$ ..... (7)

The point 3 is related to the handling of double word product $(a \cdot X_{i-1})$ in Eq. (4).

Let $a = \pm 2^{k1} \pm 2^{k2}$, $m = 2^p - 1$.

$x_i \equiv a \cdot x_{i-1} \equiv x_{i-1} (\pm 2^{k1} \pm 2^{k2}) \equiv \pm 2^{k1} \cdot x_{i-1} \pm 2^{k2} \cdot x_{i-1}$

$\equiv \pm (2^{k1} \cdot x_{i-1} \bmod m) \pm (2^{k2} \cdot x_{i-1} \bmod m) \pmod m$.

When $m = 2^p - 1$, multiplication by $2^k \pmod m$ can be obtained by swapping the high-order and lo -order bits

Let x be a p-bit integer, $k < p$.

$2^k \cdot x \bmod (2^p - 1) = x$ (high-order k bits) + x (low-order $p - k$ bits) $\cdot 2^k$.

This action can even be implemented in hardware with n logic gates but wires.

The point 4 needs further discussion. Consider $L = 2$. When prime $p \equiv 3 \pmod 4$, $A^{(p-1)/2} \equiv 1 \pmod p$, A has two square roots $\pm A^{(p+1)/4} \pmod p$. When $m = 2^N$, $N \geq 3$, $A \equiv 1 \pmod 8$, and a is a square root of A, the other three roots are $-a$, $a + 2^{N-1}$, and $-a + 2^{N-1}$. One of these roots can be obtained from the a_N of the following recursive equation [2, p.141]:

$a_3 = 1$, $a_{t+1} \equiv a_t + (a_t^2 - A)/2 \pmod m$ ..... (8)

3. Example Multiplication-Free Generators

Table I lists some of linear congruential generators using this technique. All of generators use $m = 2^{32}$ or $p = 2^{31} - 1$. The technique is also applicable to moduli such as 2^{64} and $2^{61} - 1$. Such large moduli will be in common use when 64-bit processors are available in most desktops and 64-bit codes are as fast as 32-bit codes.

Generators G1 and G2 are for Eq. (3); G3 and G4 are for Eq. (4); G5 is for Eq. (6). G2 and G4 are parallel versions with $L = 2$ for Eqs.(3) and (4), respectively. The right columns show spectral test results of these generators. The notation here follows that of Anderson [1]. The ν_t is the "wave numbers" in t-dimensional space. The $1/\nu_t$ provides a measure of the granularity of a generator in t-dimensional space. In theory, $\nu_t \leq \gamma_t \cdot m^{1/t}$, $q_t = \nu_t / (\gamma_t \cdot m^{1/t})$ represents a relative quality measure. The constant γ_t can be found in [8, p.105]. The $\beta_t = \log_2(\nu_t)$ represents the number of random bits when t-tuples are considered. Generators G1 to G4 are selected based on q_t, $2 \leq t \leq 6$, by an exhaustive search. All these generators have $q_t > 0.7$, $2 \leq t \leq 4$. They are acceptable in comparison with the best multipliers from 3, 4], which have $q_t > 0.8$, $2 \leq t \leq 6$.

We exhaustively search generators of Eq. (6) with $n = 8$, $a_n = 2^{k1} \pm 2^{k2}$ and $a_d = 2^k$. The high-order spectral test [8, Ex. 3.3.4-24] is used to rate these generators. Because q_t is not dire tly applicable to the high-order spectral test, G5 is selected in according t ν_t. The ν_t of G5, $t \leq 8$, is constant m. The degree of polynomial $n = 8$, because it is the largest n we are able to obtain the factorization o $r = (p^n - 1) / (p - 1)$ [8, p.29]. Table II lists some factorizations of r for $p = 2^{31} - 1$ and $p = 2^{61} - 1$. Complete factorizations for $p = 2^{31} - 1$ and $n = 2..6$ can be found in 9]. Some factorizations of r are difficult to obtain even if $n < 8$. The following equation (also indicated in [9]) can greatly simplify the factorization of r when n is even:

$(p^n - 1) / (p - 1) = (p^{n/2} + 1) (p^{n/2} - 1) / (p - 1)$.

If n is a power of 2, this equation can be applied several times as shown in Table II.

Table I. The parameters and the spectral test results of five example linear congruential generators.

	Parameters	Spectral test results			
		t	q_t	β_t	v_t
G1:	$X_i = a \cdot X_{i-1} \bmod 2^N$	2	0.9161	15.98	64517
	$N = 32$	3	0.9295	10.73	1696
	$a \equiv 64517$	4	0.7621	7.86	232
	$\equiv 2^{16} - 2^{10} + 5 \pmod{2^N}$	5	0.7502	6.29	78
		6	0.7304	5.25	38
G2:	$X_i = a \cdot X_{i-1} \bmod 2^N$	2	0.9137	15.97	64347
	$N = 32$	3	0.7865	10.49	1435
	$a^2 \equiv A \pmod{2^N}$	4	0.7489	7.83	228
	$a \equiv 2655201001 \pmod{2^N}$	5	0.7118	6.21	74
	$A \equiv 4292870161 \equiv -2^{21} + 2^4 + 1 \pmod{2^N}$	6	0.6535	5.09	34
G3:	$X_i = a \cdot X_{i-1} \bmod p$	2	0.7178	15.13	35746
	$p = 2^{31} - 1$	3	0.7078	10.00	1025
	$a \equiv 2146942975$	4	0.7227	7.53	185
	$\equiv -2^{19} - 2^{14} \pmod{p}$	5	0.6077	5.78	55
		6	0.6256	4.86	29
G4:	$X_i = a \cdot X_{i-1} \bmod p$	2	0.8634	15.39	42995
	$p = 2^{31} - 1$	3	0.8694	10.30	1259
	$a^2 \equiv A \pmod{p}$	4	0.7383	7.56	189
	$a \equiv 2049892995$	5	0.7844	6.15	71
	$A \equiv 2162688 \equiv 2^{21} + 2^{16} \pmod{p}$	6	0.7119	5.04	33
G5:	$X_i = (a_d \cdot X_{i-d} + a_n \cdot X_{i-n}) \bmod p$	9	n.a.	20.00	1048584
		10	n.a.	20.00	1048584
	$p = 2^{31} - 1$, $n = 8$, $d = 3$	11	n.a.	20.00	1048584
	$f(x) = x^8 + 2048\,x^3 + 2138832895$	12	n.a.	18.06	272451
	$a_d \equiv -2048 \equiv -2^{11} \pmod{p}$				
	$a_n \equiv -2138832895 \equiv 8650752 \equiv 2^{23} + 2^{18} \pmod{p}$				

n.a. = not available

Table II. Some factorizations o $r = (p^n - 1) / (p - 1)$ for $p = 2^{31} - 1$ and $p = 2^{61} - 1$.

p	n	r
$2^{31} - 1$	2	2^{31}
$2^{31} - 1$	4	$2^{32} \times 5 \times 733 \times 1709 \times 368140581013$
$2^{31} - 1$	8	$2^{33} \times 5 \times 17 \times 41 \times 733 \times 1709 \times 21529 \times 368140581013 \times 7086516946227271152326737246 57$
$2^{61} - 1$	2	2^{61}
$2^{61} - 1$	4	$2^{62} \times 2113 \times 3389 \times 91961 \times 4036962584010807014809213$
$2^{61} - 1$	8	$2^{63} \times 337 \times 2113 \times 3389 \times 91961 \times 4036962584010807014809213 \times 4194295702738004335948501884892915848 3 \times 1960079822522112679733601197034 73$

4. The Implementation

This section presents the source codes for generators G1, G3, and G5. These programs mainly use adds, subtracts, and shifts, and use no conditional statements. The hardware design of G3 is also presented. Figure 1 is the C program for G1. The multiplication is replaced with 3 shifts and 3 adds/subtracts.

Figure 2 is the C program for G3. LOG_W is the number of bits of integer type (int). LOG_M = $\lceil \log_2 m \rceil$. Variable x0 is declared as "unsigned int", thus all the right shifts of x0 will fill 0 from the left. When x1 < 0, the sign bit of x1 is 1, i.e., x1 = -2^{31} + (x1 & M). Thus, x1 ≡ M - 2^{31} + (x1 & M) ≡ (x1 & M) - 1 (mod M). Since x1 is a 32-bit unsigned integer, (x1 >> 31) obtains its sign bit. Directl using x1's sign bit in computation can avoid the conditional statement on the sign bit. This makes the program easy to vectorize. This is a slight improvement to the code in [15].

Figure 3 is the C program for G5. N denotes n, and D denotes d in Eq. (6). The seed is an integer array of size N. Assume that N is a power of 2. Index wrapping on the arra seed is by an bitwise-and (&) with N - 1. The sign bit of x1 is used as a carry bit for the statement "x1 = w1 + w2" and as a borrow bit for the statement "x1 = x1 - w3".

Figure 4 shows the logic design of G3. $\overline{X}$ is one's complement of X. The numbers in the brackets of $\overline{X}$ represent the bits of $\overline{X}$. We obtain

$-2^{19} \cdot X = \overline{X}[11..0; 31..12]$; and

$-2^{14} \cdot X = \overline{X}[16..0; 31..17]$.

This is equivalent to replacing part of the C code in Figure 2 by the following statements:

```
x1 = (~w1 & M) + (~w2 & M);
seed = (x1 & M) + (x1 >> LOG_M);
```

The operator '~' is one's complement. The adder has two 31-bit inputs. The carry-out bit feeds back to the carr -in bit of the adder. The result of the adder is the next random number X'.

This design is very simple in comparison with those in [5, 11]. Note that the method takin $a \cdot x \pmod{p}$ using no division was covered in the claim 2 of U.S. patent No. 5,317,528 [5]. As addressed in Introduction, the division-free technique is a prior art [12].

```
/* m = 2^32, a = 64517 = 2^16 - 2^10 + 5 */
#define K1    16
#define K2    10
static unsigned int seed = 1;
void set_seed(unsigned int s)
{
  if (s>0) seed = s;
}
unsigned int get_rand()
{
  unsigned int x0 = seed;
  seed = (x0<<K1) - (x0<<K2) + (x0<<2) + x0;
  return seed;
}
```

Figure 1. The C code for generator G1.

```
/*  m=2^31-1, a=2146942975=- 2^19 - 2^14 */
#define LOG_W 32
#define LOG_M 31
#define M     0x7fffffff
#define K1     19
#define K2     14
static int seed = 1;
void set_seed(unsigned int s)
{
  if (s>0)  seed = s;
}
unsigned int get_rand()
{
  unsigned int x0 = seed;
  unsigned int x1;
  int w1, w2;

  w1 = (x0>>(LOG_M-K1)) /*low-order bits */
   + ( (x0<<(K1+LOG_W-LOG_M)) >>
(LOG_W-LOG_M) ); /* high-order*/

  w2 = (x0>>(LOG_M-K2)) /* low-order bits */
   + ( (x0<<(K2+LOG_W-LOG_M)) >>
(LOG_W-LOG_M) ); /* high-order*/
  x1 = M - w1 - w2;
  seed = (x1 & M) - (x1 >> LOG_M);
  return seed;
}
```

Figure 2. The C code for generator G3.

```
/* m = 2^31 -1
   X(i) = a_d * X(i-3) + a_n * X(i-8)
   a_n = 2^23 + 2^18;  a_d = - 2^11 */
#define N     8
#define D     3
#define LOG_W 32
#define LOG_M 31
#define M     0x7fffffff
#define K1    23
#define K2    18
#define K3    11
static int seed[N];
static int count=0;
void set_seed(unsigned int s)
{
  int i;
  if (s > 0) seed[0] = s;
  for(i=1; i<N; i++)
    seed[i] = 0;
}
unsigned int get_rand()
{
  unsigned int x0 = seed[count];
  unsigned int xd = seed[(count-D) & (N-1)];
  unsigned int x1;
  int w1, w2, w3;

  w1 = (x0 >>(LOG_M-K1)) /*low-order bits */
   + ( (x0<<(K1+LOG_W-LOG_M)) >>
(LOG_W-LOG_M) ); /* high-order bits */
  w2 = (x0>>(LOG_M-K2)) /* low-order bits */
   + ( (x0<<(K2+LOG_W-LOG_M)) >>
(LOG_W-LOG_M) ); /* high-order bits */
  w3 = (xd >>(LOG_M-K3)) /*low-order bits */
   + ( (xd<<(K3+LOG_W-LOG_M)) >>
(LOG_W-LOG_M) ); /* high-order bits */
  x1 = w1 + w2;
  x1 = (x1&M) + (x1>>LOG_M); /*add carry */
  x1 = x1 - w3;
  x1 = (x1&M)-(x1>>LOG_M); /*subtract borrow
*/
  seed[count] = x1;
  count = (count+1) & (N-1);
  return x1;
}
```

Figure 3. The C code for generator G5.

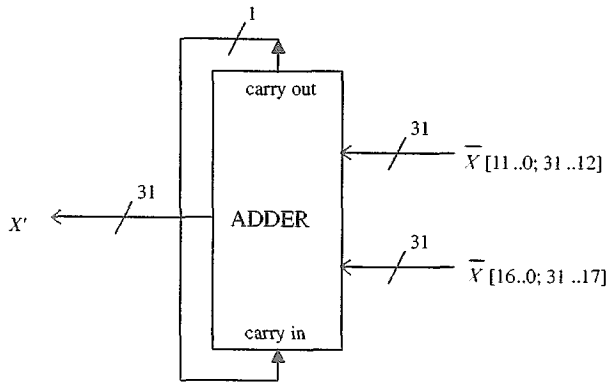

Figure 4. The logic design for G3.

5. Statistical and Timing Results

We apply three empirical tests to generators G1 to G5. The tests follow Wu [15]. The frequency test takes x mod 12. We take (x >> 6) mod 12 for G1 and G2, because their low-order bits are not very random. The run test examines the length of "run up" sequences, categorized into [1, 6] and >6. The maximum-of-t test takes the maximum of 5 consecutive numbers x_i and examines whether $max(x_i) < 7/8$ m. The degrees of freedom of these tests are 11, 6, and 1, respectively. We use the chi-square test in 6 rounds. Each test consumes 2,000,000 random numbers. The 36,000,000 consecutive random numbers are generated by seed 1. The chi-square result V is rejected if V is outside [1%, 99%]. The result is "suspect" i V is in [1%, 5%] or [95%, 99%]. The result is "almost suspect" if V is in [5%, 10%] or [90%, 95%]. The results are shown in Table III. All these generators are satisfactory.

Table IV shows the timing results of G1 to G5. We compared the performance of multiplication-free (mul-free) development with the typical multiplication (mul) method. For the mul versions of G3 to G5, we use 64-bit integers ("long long int") to handle the double word product and use shifts and adds for the division. All these programs generate an array of random numbers in one procedure call. A test was conducted that generated 50,000,000 random integers in 1,000 calls. The code on SparcStation was compiled by GNU CC (gcc -O); the code on PC was compiled by djgpp, D.J. Delorie's port of GNU CC. The codes in Alpha and RS6K were compiled by the vendor's C compiler (c -O). The execution times were measured in seconds by the clock function. All programs were executed five times and the minimum ones were taken.

The speedups on G1 and G2 would be very small, because about 3 shifts and 3 adds are needed to replace one multiplication. It is surprising that we get a speedup of 1.30 in Alpha. The speedups on G2 are mainly due to the longer time in multiplying b $a = 2655201001$. The speedups on G3 to G5 are as expected except the dramatic speedups 7.57 and 10.24 obtained on SparcStation. The actual speedups on G3 to G5 will be less if the mul versions are coded in assembly.

6. Conclusion and Future Work

This paper has presented the design and implementation of multiplication-free linear congruential generators. The statistical and timing results of five example generators G1 to G5 have been presented. Our result shows that multiplication-free linear congruential generators can achieve high speed with no degradation on quality. Vectorization of these generators using multimedia extension instructions has been addressed in brief. The hardware design of generator G3 has also been presented. The future work on these generators includes: (1) further theoretical and experimental analyses; 2) hardware implementations for applications such as test patterns generation of sel -testing integrated circuits; and 3) applying to very large moduli such as 2^{128} and $2^{127} - 1$, which can be represented as a double word in 64-bit machines.

Table III. Statistical results for generators G1 to G5.

	Frequency test		Run test		Maximum-of-t	
G1	6.7158		7.0171		0.1427	
	11.6529		7.9178		0.4479	
	1.1787	R	5.7856		1.0412	
	15.8069		8.8324		1.0597	
	9.3125		11.3685	AS	3.3134	AS
	6.0053		9.9845		0.8852	
G2	8.9051		4.5279		0.0404	
	17.5204	AS	2.5835		0.0498	
	12.6557		3.7965		0.5881	
	9.0263		11.3234	AS	1.4570	
	14.2090		4.9657		0.0950	
	7.8747		9.4397		0.1053	
G3	10.7834		8.0457		2.5578	
	13.1993		1.6961	S	0.0390	
	18.9748	AS	12.5857	AS	1.3155	
	11.3936		8.2869		0.2946	
	19.1859	AS	4.8435		3.8271	AS
	3.7377	S	6.6850		0.2485	
G4	6.1138		6.5518		2.1227	
	6.3843		4.7412		0.7757	
	11.8161		4.9098		0.4305	
	9.0772		2.0365	AS	1.5109	
	11.0775		15.0830	S	0.1451	
	15.1131		3.3433		0.2360	
G5	11.3233		1.0765	S	0.5085	
	12.1466		13.3719	S	0.0747	
	29.9432	R	6.7521		0.0379	
	12.0196		5.7374		0.2239	
	16.4973		2.4928		0.6427	
	11.9700		7.3736		0.0007	S

R=Reject; S=Suspect; AS=Almost Suspect.

Table IV. Timing results of generators G1 to G5 on four platforms.

Generator		PC Pentium-90	Sparc Station-10	RS6K/ 590	Alpha 3K/500
G1	mul-free	6.98 s.	11.37 s.	1.84 s.	1.28 s.
	mul	6.98 s.	11.35 s.	1.07 s.	1.67 s.
	speedup	1.00	1.00	0.58	1.30
G2	mul-free	6.92 s.	12.18 s.	1.78 s.	1.30 s.
	mul	6.92 s.	59.38 s.	1.10 s.	4.05 s.
	speedup	1.00	4.88	0.62	3.12
G3	mul-free	9.61 s.	16.30 s.	4.24 s.	3.68 s.
	mul	20.82 s.	27.75 s.	7.53 s.	5.70 s.
	speedup	2.17	1.70	1.78	1.55
G4	mul-free	10.22 s.	18.57 s.	3.52 s.	3.87 s.
	mul	20.71 s.	140.58 s.	7.52 s.	7.73 s.
	speedup	2.03	7.57	2.14	2.00
G5	mul-free	21.42 s.	26.40 s.	5.53 s.	6.88 s.
	mul	60.48 s.	270.34 s.	14.28 s.	14.32 s.
	speedup	2.82	10.24	2.58	2.08

REFERENCES

[1] Anderson, S. L. (1990), "Random Number Generators on Vector Supercomputers and Other Advanced Architectures," *SIAM Review*, Vol. 32, No. 2, pp. 221-251.

[2] Anglin, W. S. (1995), *The Queen of Mathematics: An Introduction to Number T heory*, Kluwer Academic Publishers.

[3] Fishman, G. S., and Moore, L. R. (1986), "A Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus $2^{31} - 1$," *SIAM J. Sci. Stat. Comput.*, Vol. 7, No. 1, pp. 24-45.

[4] Fishman, G. S. (1990), "Multiplicative Congruential Random Number Generators with Modulus 2^{β}: An Exhaustive Analysis for β=32 and a Partial Analysis for β=48," *Mathematics of Computatio* , Vol. 54, No. 189, pp. 331-344.

[5] Gofman, E. (1994), "Random Number Generator", U.S. Patent No. 5,317,528.

[6] Kao, C., and Tang, H.-C. (1997), "Upper Bounds in Spectral Test for Multiple Recursive Random Number Generators with Missing Terms," *Computers Math. Applic.*, Vol. 33, No. 4, pp. 119-125.

[7] Kernighan, B.W., and Ritchie, D. M. (1988), *The C Programming Language*, 2nd Ed., Prentice-Hall.

[8] Knuth, D. E. (1981), *The Art of Computer Programming Vol 2: Seminumerical Algorithms*, 2nd ed., Addiso -Wesley, MA.

[9] L'Ecuyer, P., Blouin, F., and Couture, R. (1993), "A Search for Good Multiple Recursive random Number Generators," *ACM Trans. on Modeling and Computer Simulation*, Vol. 3, No. 2, pp. 87-98.

[10] Park, S.K., and Miller, K. W. (1988), "Rando Number Generators: Good Ones Are Hard to Find," *Communications of the ACM*, Vol. 31, No. 10, pp. 1192-1201.

[11] Paplin 'ski, A.P., and Bhattacharjee, N. (1996) "Hardware Implementation of the Lehmer Random Number Generator," *IEE Proc.-Comput. Digit. Tech.*, Vol. 143, No. 1, pp. 93-95.

[12] Payne, W. H., Rabung, J. R., and Bogyo, T. P. (1969), "Coding the Lehmer Pseudo-random Number Generator," *Communications of the ACM*, Vol. 12, No. 2, pp. 8 -86.

[13] Peleg, A., Wilkie, S., and Weiser, U. (1997), "Intel MMX for Multimedia PCs," *Communications of the ACM*, Vol. 40, No. 1, pp. 24-38.

[14] Schrage, L. (1979), "A More Portable Fortran Random Number Generator," *ACM Trans. on Mathematical Software*, Vol. 5, No. 2, pp. 132-138.

[15] Wu, P.-C. (1997), "Multiplicative, Congruentia Random Number Generators with Multiplier $\pm 2^{t_1} \pm 2^{t_2}$ and Modulus $2^p - 1$," *ACM Trans. on Mathematical Software*, Vol. 23, No.2, pp.255-265.