A SHIFT-REGISTER SEQUENCE RANDOM NUMBER GENERATOR IMPLEMENTED ON THE MICROCOMPUTERS WITH 8088/8086 AND 8087 *

Ting-Wai CHIU and Tian-Shin GUU

Physics Department, National Taiwan University, Taipei, Taiwan 10764

Received 21 January 1987; in revised form 1 April 1987

A shift-register sequence random number generator is programmed in Intel 8088/8086 and 8087 assembly language. Its performance is good in terms of its speed, long period and consistently producing reliable results in extensive statistical tests. This generator may promote the microcomputers with 8088/8086 and 8087 to become a workstation for developing programs for Monte Carlo simulations of lattice field theories.

PROGRAM SUMMARY

Title of program: TRCG

Catalogue number: AAXI

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Computer: IBM-PC/XT/AT and compatibles

Operating system: PC (MS) DOS 2.0 or later

Programming language used: the subroutines are written in 8086/87 assembly language. Having been assembled by the Macro Assembler, they can be linked to the TURBO-87 PASCAL version 3.x or later

No. of bits in a word: 16

Peripherals used: color monitor display, printer

Number of lines in combined program and test deck: 500

Keywords: uniform random number generator, Monte Carlo simulation, stochastic quantization, lattice field theories, spin models

Nature of the physical problem

The subroutines generate a sequence of uniformly distributed real and integer pseudo random numbers with the maximum period of $2^{*}250-1$. They can be used in Monte Carlo simulations of many different physical problems.

Typical running time

The actual running time depends on the system clock of the microcomputer as well as the compiler used. For a IBM-PC compatible with 4.77 MHz CPU 8088/8087 and the test program compiled by the TURBO-87 PASCAL, the speeds of the single precision subroutines and functions are:

- (i) The seeding subroutine (SEED) takes 53 ms.
- (ii) The uniform real number generator (TRCG) generates 4140 uniformly distributed random numbers in the interval $[0,1]$ within one second.
- (iii) The integer random number generator (ITRCG) generates 6350 random integers in one second.

Unusual features of the program

The program generates the pseudo random numbers using the Tausworth's shift register sequence algorithm [1], which has a very long maximum period. In our programs, we choose the maximum period to be $2^{*}250-1$. However, the period can be easily programmed to be much longer.

Reference

- [1] R.C. Tausworth, Math. Comput. 19 (1965) 201.

* Research is supported in part by National Science Council, Taiwan.

LONG WRITE-UP

1. Introduction

There are many situations arisen in various fields for which the mathematical model calls for long sequences of pseudo random numbers. An important example in the field of computational physics is the Monte Carlo simulation [1] of lattice field theories [2].

The results of these computer experiments depend crucially on the quality (randomness) of the sequences of pseudo random numbers. Therefore, it is of paramount importance to implement a good random number generator (RNG) before one attempts the Monte Carlo simulation on any computer system. Furthermore it would be advantageous in a single system to have several different random number algorithms, to be eventually mixed in order to test the independence of the result from the algorithm. With the advance of the computer technology, the microcomputers are now widely available at offices and at home. For those equipped with the microprocessor 8088 and the floating point coprocessor 8087, their number crunching capability has attained 5% of the super-mini VAX 11/780. They would provide an excellent software development station for the Monte Carlo simulations of the Lattice Field Theories as well as other systems. Unfortunately, the random number generators available for these microcomputers are exclusively employing the old-fashioned linear congruential method which has the drawbacks of short cycle length and strong short-range correlations. To create a better environment for developing software for Lattice Field Theories, we implement the Shift Register Sequence Random number (TRNG) generator which is programmed in 8088/8087 assembly language, and should be transportable to any microcomputer equipped with 8088/8087 (i.e., the IBM/PC compatibles). The TRNG was introduced by Tausworthe [3] in 1965 and programmed in ASSEMBLER on an IBM 370/168 by Kirkpatrick and Stoll [4]. One major advantage of TRNG is its effectively unlimited cycle length which is essential for Monte Carlo simulations of Lattice Field

Theories. The outline of this paper is as follows. In section 2, we briefly review the algorithms for the random number generators. In section 3, we describe our implementation of the TRNG in 8088/8087 assembly language. In section 4, we perform several tests on the randomness of this TRNG. In section 5, we conclude and discuss.

2. Algorithms

The most widely used class of pseudo random numbers generators was suggested by Lehmer in 1949 [5] and is known as the linear congruential random numbers generator. The algorithm can be described by the recursive formula

$$x_{i+1} = ax_i + c \pmod{m}, \quad (1)$$

where $\{x_i\}$ is the integer sequence generated, a and c are some carefully chosen constants. The linear congruential sequence has the maximum period m if and only if the following holds [6]: (i) $(a - 1)$ is a multiple of 4 if m is a multiple of 4; (ii) $(a - 1)$ is a multiple of p , for every prime p dividing m ; (iii) c is a relatively prime to m . Usually the additive constant c is set to zero and the generator is called "multiplicative congruential". In this case, the maximum period m is achieved if m is prime and a is a positive prime root of m , i.e., $a^m = 1 \pmod{m}$ but $a^n \neq 1 \pmod{m}$ for all $n < m$. The integer sequence $\{x_i\}$ can be converted into floating point numbers in the interval $(0, 1)$ by $u_i = x_i/m$. The choice for the modulus m having been dictated by the computer word length. For computers with 32 bit words, the Mersenne prime $m = 2^{31} - 1$ is a convenient modulus. Many subroutines [7] using $a = 7^5 = 16807$, which is primitive with respect to $2^{31} - 1$, have been extensively used. For computers with longer word length (e.g. CDC CYBER 170 or CDC CYBER 205), (≥ 60 bits), the prime $m = 2^{47} - 1$ is chosen. This RNG (function RANF( )) is implemented in all CDC computers and has been extensively used for the Monte Carlo simulations of the Lattice Field Theories [8]. Although these multiplicative congruential RNG were found to have satisfactory statistical properties (which has relatively strong short range correlations, but does not

have long-range correlations), and is quite fast, and actually reach their maximal period, their drawback is that their period is too short for any serious Monte Carlo simulations of Lattice Field Theories or statistical systems of moderate size [9] (e.g. 3D Ising model of size $128 \times 128 \times 128$).

An algorithm for producing pseudo random bit sequences of effectively unlimited period was proposed by Tausworthe [3] in 1965. Let $a = \{a_n\}$ be the sequence of bits (0 or 1) generated by the linear recursion formula

$$a_k = \sum_{i=1}^{m-1} c_i a_{k-i} + a_{k-m} \pmod{2} \quad (2)$$

for any given set of integers c_i ($i = 1, 2, \dots, m-1$), each having the value 0 or 1. Since a_k is determined solely from the previous m -tuple $(a_{k-1}, \dots, a_{k-m})$, thus the maximum possible period is $(2^m - 1)$. This is achieved if and only if the polynomial

$$f(x) = 1 + \sum_{i=1}^m c_i x^i \quad (3)$$

is primitive over the GF(2) [10]. This algorithm was extended [11] to generate a sequence of n -bit random integers since the n -bit integer can be regarded as n columns of random bits. Thus the algorithm

$$x_k = c_1 x_{k-1} \oplus \dots \oplus c_{p-1} x_{k-p+1} \oplus x_{k-p}, \quad (4)$$

where $\oplus$ is the “bit by bit exclusive or” operation, may generate very long pseudo random integer sequence. The simplest case of algorithm (4) is the following,

$$x_k = x_{k-q} \oplus x_{k-p}. \quad (5)$$

One would intuitively expect the best result from (5) if the degree p is a Mersenne exponent and $2^p - 1$ is a Mersenne prime. A table of primitive trinomials whose degree is a Mersenne exponent was published by Zierler [12]. Up to now, all TRNG were implemented by using the simplest algorithm (5). One problem of the algorithm (5) is that it requires careful initialization. Lewis and Payne [11] proposed initializing $\{x_1 \dots x_p\}$ such that each column of bits contains a delayed rep-

lica of the same basic bit sequence, using a delay of order $100p$ steps between columns. Kirkpatrick and Stoll [4] proposed simply using a good random number generator of multiplicative congruential type to produce the first p integers. In this paper, we use a slightly different procedure to generate the initial p integers. The details are described in the next section.

3. Implementation of the TRNG

We chose the primitive trinomial with $q = 103$ and $p = 250$. However, our implementation can be easily extended to other primitive polynomials. First of all, we use the multiplicative congruential algorithm

$$x_{i+1} = 16807x_i \pmod{2^{31} - 1}$$

to generate a sequence of integers x_i ; $0 \leq i \leq 15$. Then we group sixteen consecutive 4 bit short-integers in the sequence to form a 64 bit (8 bytes) long-integer. The initial 250 long-integer seeds are generated in this manner. To ensure the initial seeds are linearly independent, we employ the following procedure to randomize the seeds. We obtain the time in 4 bytes (hour, minute, second and milliseconds) from the internal clock of the microcomputer, then multiply it with 16807 (mod $2^{63} - 1$). This number is then used to “bit by bit exclusive or” with each one of the seeds. The resulting values are then the new seeds. We find that this randomization provides a very good initialization for the TRNG. Programming the algorithm (5) for a given p and q is not difficult. To optimize the speed, we have taken advantage of the concurrency of 8088 and 8087. While the 8088 is performing the “bit by bit exclusive or” action, the 8087 is normalizing the 8-bytes integer resulting from the previous “exclusive or” action to a double precision real number. Statistical tests of TRCG will be described in the next section. We have also implemented a two-byte integer RNG based on the following algorithm:

$$x_{n+p} = x_n + x_{n+q}, \pmod{m}, \quad (6)$$

where p and q are chosen to be 250 and 147, respectively. The modulus m is set to be the upper

Table 1
Speed test of various random number generators

Generators	TRCG	ITRCG	TURBO-PASCAL real	TURBO-PASCAL integer	MS-BASIC Compiler real
no./s	4140	6350	4000	6300	2000

bound of the random integer sequence. Variants of this algorithm have been advocated by Knuth [13]. The virtues of this type of RNG are its speed, long period and consistently producing reliable results in extensive statistical tests. The initial 250 seeds are generated by the same procedure described before, though only the lowest order two bytes are used. Statistical test of ITRCG will be described in the next section.

We compare the speeds of our random number generators to those implemented in the most widely used compilers. We pick the real RNG (intrinsic function RND) implemented in the MS-BASIC Compiler [14], and the real RNG and integer RNG in Turbo Pascal v3.0 [15].

Our results are listed in table 1. The tests were carried out in a microcomputer with CPU (8088, 4.7 MHz) and floating point coprocessor (8087, 5 MHz).

The maximum possible period of TRCG and ITRCG is $2^{250} - 1$. It is too large to be detected in any system within a reasonable time. Our test has shown that their period is bigger than 2^{34} .

4. Statistical tests

There is almost no end to the number of tests that can be conceived to test the randomness of a random number generator. We shall here only performed those tests that have proved to be most useful. To see if the sequence $\{x_i\}$ is independently and uniformly distributed, uniformity and independence should be tested separately. A simple but powerful test for both of these qualities is to plot (by the computer itself) pairs of consecutive random numbers on the screen of the monitor. During the process of filling up the screen by these points, we can visually detect any striated density variations due to non-uniformity or self-correlations of the random number sequence. We

found that TRCG and ITRCG both filled the 480–200 color graphic screen with about 1 300 000 points (pairs), and there is no any striated density variation in any intermediate “pictures”. On the other hand, the real RNG in Turbo Pascal v3.0 looks to be uniform and homogenous after 5 000 points has been plotted (fig. 1). However, it turns out to be disastrous after 570 000 points have been plotted, strong evidence of striated density variations has shown up completely (fig. 2). In fact, the empty portions of the screen will never be filled up no matter how many points are plotted. The integer RNG in Turbo Pascal v3.0 as well as the real RNG in MS-BASIC compiler have passed this visual test and filled the full screen with 1 200 000 pairs, without any “patterns” appearing in any intermediate “pictures”. We could imagine that a three dimensional plot of triples of consecutive random numbers with simultaneous two di-

Fig. 1. 10000 random numbers distributed in the unit interval generated by real random number generator in Turbo-Pascal v3.0 and plotted as 5000 (x, y) pairs.

Fig. 2. More than 1000000 random numbers generated by Turbo-Pascal v3.0 and plotted as (x, y) pairs.

mensional cross-sectional display at various planes and orientations should be a powerful and stringent test for the randomness of the random number generators.

To examine the uniformity quantitatively, we perform the chi-square test and the Kolmogorov–Smirnov (KS) test. For the chi-square test, we measure the statistic

$$\chi^2 = \sum_{1 \leq i \leq k} (T_i - np_i)^2 / np_i, \quad (7)$$

where p_i is the probability that each observation falls into category i , T_i is the number of observations that actually do fall in category i , and n is the total number of independent observations. We

choose the total number of categories to be 50; that is, to divide the interval $(0, 1)$ into 50 subintervals for the real RNG, and put the upper bound (or the modulus m) in the integer RNG to be 50. If the total numbers of independent observations n is large enough, the χ^2 value will be less than or equal to some value y with approximate probability p . For degrees of freedom $\nu \equiv k - 1$ greater than 30, the theoretical approximate formula for the value y [16] is

$$y \approx \nu \left(1 - 2/9\nu + x_p \sqrt{2/9\nu} \right)^3, \quad (8)$$

where $p = 1 - P(x_p)$ and P is the chi-square probability function. We did 2000 observations in a single sampling and obtained the empirical χ^2 value by eq. (7), then in 5000 samplings we determined the probability that the empirical χ^2 value to be less than or equal to y in eq. (8). Our results are listed in table 2. It appears that every RNG has χ^2 distributions close to the theoretical probability, even the real RNG in Turbo-Pascal does not show any sign of non-uniformity. However, we have already known that it is defective (see fig. 2). This indicates that the visual (spectral) test is a better test than the “one-dimensional” χ^2 distribution test.

The next test on the uniformity is the Kolmogorov–Smirnov (KS) test. We measure the statistics

$$K_n^+ = \sqrt{n} \max_{1 \leq i \leq n} (n_i/n - F(x_i)), \quad (9a)$$

$$K_n^- = \sqrt{n} \max_{1 \leq i \leq n} (F(x_i) - (n_i - 1)/n), \quad (9b)$$

where n is the total number of observations, $\{x_i, i = 1, \dots, n\}$ is the random number sequence observed, n_i is the number of random numbers less

Table 2

Chi-Square Distribution Statistics (the number of degrees of freedom is 49. 5000 samplings are performed with 2000 random numbers in each)

y_p	28.893	33.967	42.012	48.336	55.265	66.275	74.985
Theoretical probability	0.01	0.05	0.25	0.50	0.75	0.95	0.99
ITRCG	0.0094	0.0430	0.2444	0.4924	0.7574	0.9428	0.9860
TRCG	0.0114	0.0520	0.2554	0.4954	0.7510	0.9506	0.9880
TURBO-PASCAL integer	0.0084	0.0522	0.2556	0.4990	0.7530	0.9498	0.9890
TURBO-PASCAL real	0.0116	0.0518	0.2532	0.4938	0.7432	0.9430	0.9894

Table 3

Kolmogorov–Smirnov Distribution Statistics (5000 samplings with 50 random numbers in each. p^- and p^+ are the probabilities that K_- and K_+ equal or less than K_p , respectively)

K_p		0.20	0.31	0.57	0.87	1.04
p		0.10	0.20	0.50	0.80	0.90
TURBO-PASCAL	p^-	0.0906	0.1970	0.4914	0.7972	0.9044
real	p^+	0.0982	0.2070	0.5052	0.8018	0.9002
TRCG	p^-	0.1020	0.1998	0.5030	0.8014	0.8962
	p^+	0.1064	0.2058	0.5060	0.8002	0.8982

than x_i , $F(x)$ is the ideal probability that the observed value is less than x . Here K_n^+ measures the greatest amount of deviation when the empirical distribution function is greater than the ideal distribution function, and K_n^- measures the maximum deviation when the empirical distribution is less than the ideal distribution. The factor $\sqrt{n}$ is used to compensate the standard deviation of empirical distribution which is proportional to $1/\sqrt{n}$, such that the standard deviation of K_n^+ and K_n^- is independent of n . For $n > 30$, the K_n^+ and K_n^- will be less than

$$K_p = \left[\frac{1}{2} \ln(1/(1-p)) \right]^{1/2} - 1/6\sqrt{n} + \mathcal{O}(1/h) \quad (10)$$

with the probability p . We did 50 observations in a single sampling and calculated the empirical K_n^+ and K_n^- values by eq. (9), then in 5000 samplings we determined the probabilities that the empirical values K_n^+ and K_n^- values to be less than K_p in eq. (10). Our empirical results are listed in table 3. Both generators show good statistics.

N -tuples of random numbers are often used in Monte Carlo simulations of Lattice Field Theories. To test the uniformity of successive triples of random numbers generated by these RNGs, $4 \times$

10^5 triples were constructed for each and assigned to cells in the unit cube, with a resolution of $20 \times 20 \times 20$ cells. We measure the chi-square statistics

$$\chi^2 = \sum_{i,j,k=1}^L (T(i, j, k) - \bar{n})^2 / \bar{n},$$

where $T(i, j, k)$ denotes the number of triples falling into the cell (i, j, k) and $\bar{n}$ is the average number of triples per cell. Since the degrees of freedom are extremely large (≈ 8000), eq. (8) implies that the empirical χ^2 value would be almost equal to ν with standard deviation $\approx 1.4\sqrt{\nu}$ if the random numbers are uniformly and independently distributed. We did 20 samplings with 4×10^5 observations in each sampling. The empirical results are listed in table 4. It is evident that the real RNG of Turbo-Pascal has unusual high χ^2 statistics. This indication is consistent with the non-uniformity displayed in fig. 2. In other words, this “three-dimensional” chi-square test is more reliable than the usual “one-dimensional” chi-square test.

To examine the independence, we perform the autocorrelation test and the run length test, respectively. For the autocorrelation test, we mea-

Table 4

Chi-Square Statistics for Triple Distribution (20 samplings with 4×10^5 triples in each. The last row gives the standard deviations)

RNG	Theoretical expectation	TRCG	ITRCG	TURBO-PASCAL		MS-BASIC COMPILER real
				real	integer	
χ^2	8000	7963 ± 100	8065 ± 146	10134 ± 143	8021 ± 92	7812 ± 159

Table 5

Autocorrelation Statistics (100 samplings with 10^4 random numbers each, mean values and standard deviations are shown)

	TRCG	TURBO-PASCAL real	MS-BASIC real
$A(0)$	$0.333 \pm 2.80 \times 10^{-3}$	$0.333 \pm 3.16 \times 10^{-3}$	$0.333 \pm 3.08 \times 10^{-3}$
$A(1)$	$0.250 \pm 2.83 \times 10^{-3}$	$0.250 \pm 3.03 \times 10^{-3}$	$0.250 \pm 3.15 \times 10^{-3}$
$A(2)$	$0.250 \pm 2.90 \times 10^{-3}$	$0.250 \pm 3.11 \times 10^{-3}$	$0.250 \pm 3.11 \times 10^{-3}$
$A(3)$	$0.250 \pm 2.80 \times 10^{-3}$	$0.250 \pm 3.20 \times 10^{-3}$	$0.250 \pm 3.17 \times 10^{-3}$
$A(4)$	$0.250 \pm 3.07 \times 10^{-3}$	$0.250 \pm 3.13 \times 10^{-3}$	$0.250 \pm 2.99 \times 10^{-3}$
$A(5)$	$0.250 \pm 2.84 \times 10^{-3}$	$0.250 \pm 3.23 \times 10^{-3}$	$0.250 \pm 2.96 \times 10^{-3}$
$A(6)$	$0.250 \pm 2.93 \times 10^{-3}$	$0.250 \pm 3.14 \times 10^{-3}$	$0.250 \pm 3.11 \times 10^{-3}$
$A(7)$	$0.250 \pm 2.83 \times 10^{-3}$	$0.250 \pm 3.12 \times 10^{-3}$	$0.250 \pm 3.11 \times 10^{-3}$
$A(8)$	$0.250 \pm 2.89 \times 10^{-3}$	$0.250 \pm 3.20 \times 10^{-3}$	$0.250 \pm 2.94 \times 10^{-3}$
$A(9)$	$0.250 \pm 2.85 \times 10^{-3}$	$0.250 \pm 3.17 \times 10^{-3}$	$0.250 \pm 3.08 \times 10^{-3}$
$A(10)$	$0.250 \pm 3.02 \times 10^{-3}$	$0.250 \pm 3.16 \times 10^{-3}$	$0.250 \pm 3.22 \times 10^{-3}$
$A(11)$	$0.250 \pm 2.98 \times 10^{-3}$	$0.250 \pm 3.27 \times 10^{-3}$	$0.250 \pm 3.08 \times 10^{-3}$
$A(12)$	$0.250 \pm 2.94 \times 10^{-3}$	$0.250 \pm 3.26 \times 10^{-3}$	$0.250 \pm 3.17 \times 10^{-3}$
$A(13)$	$0.250 \pm 2.80 \times 10^{-3}$	$0.250 \pm 3.10 \times 10^{-3}$	$0.250 \pm 3.22 \times 10^{-3}$
$A(14)$	$0.250 \pm 2.90 \times 10^{-3}$	$0.250 \pm 3.24 \times 10^{-3}$	$0.250 \pm 3.06 \times 10^{-3}$
$A(15)$	$0.250 \pm 2.84 \times 10^{-3}$	$0.250 \pm 3.06 \times 10^{-3}$	$0.250 \pm 3.08 \times 10^{-3}$
$A(16)$	$0.250 \pm 2.76 \times 10^{-3}$	$0.250 \pm 3.11 \times 10^{-3}$	$0.250 \pm 3.12 \times 10^{-3}$
$A(17)$	$0.250 \pm 2.87 \times 10^{-3}$	$0.250 \pm 3.14 \times 10^{-3}$	$0.250 \pm 3.17 \times 10^{-3}$
$A(18)$	$0.250 \pm 2.86 \times 10^{-3}$	$0.250 \pm 3.13 \times 10^{-3}$	$0.250 \pm 3.14 \times 10^{-3}$
$A(19)$	$0.250 \pm 2.98 \times 10^{-3}$	$0.250 \pm 3.04 \times 10^{-3}$	$0.250 \pm 2.99 \times 10^{-3}$
$A(20)$	$0.250 \pm 3.11 \times 10^{-3}$	$0.250 \pm 2.97 \times 10^{-3}$	$0.250 \pm 3.06 \times 10^{-3}$

sure the n th autocorrelation coefficient, $A(n)$,

$$A(n) = \langle x_i x_{i-n} \rangle \quad (11)$$

for sequences of 10^4 random numbers and $0 \leq n \leq 20$. For independent distributions, $A(n)$ should be ≈ 0.25 for $n > 1$. The standard deviation expected when $A(n)$ is measured for a sequence of N numbers is

$$\Delta A(n) = \left[\langle A(n)^2 \rangle - \langle A(n) \rangle^2 \right]^{1/2} \approx 0.3N^{-1/2}. \quad (12)$$

In table 5, we show averages of $A(n)$ over 100 samples of 10^4 random numbers each together with the standard deviation $\Delta A(n)$. The results in table 5 are in good agreement with the expected value except the $\langle \Delta A(n) \rangle$ by real RNG of TURBO-PASCAL is somewhat larger than eq. (12). We also note that there is no dependence of either $\langle A(n) \rangle$ or its standard deviation on the separation, n .

To examine the higher order correlations not detected in the two point correlation $A(n)$, we measure the probability of runs of increasing or decreasing x_i . We consider a sequence such that

$x_1 < x_2 > x_3 > x_4 > \dots > x_l < x_{l+1}$ contains a run of length $(l-1)$ between x_1 and x_{l+1} . The expected number of runs of length l (including running up and running down cases) in a sequence of N independent random numbers in [17]

$$n(l) = 2N \left[(l+1)^2 + l \right] / (l+3)! \quad (13)$$

The total number of runs of all lengths is about $2N/3$. Results for 20 samplings each with $N = 10^5$ are compared in table 6 with the expected values. The observed run lengths in all RNGs agree quite well with the expected values except the real RNG in Turbo-Pascal. The total number of runs as well as the $n(1)$ and $n(2)$ in this case are all having large amount of deviation from the corresponding expected values. This statistics is consistent with the two dimensional plot in fig. 2 as well as the triple distribution statistics.

5. Conclusion and discussion

In this paper, a fast shift-register sequence random number generator (TRCG) has implemented

for the microcomputers with CPU 8088/8087. The programs are transportable to any microcomputers with 8088/8087 or 8086/8087, and are ready to link with the Turbo-Pascal v3.0. Because the code segment and data segment are separated from each other in most compilers (e.g. Professional Fortran), modifications have to be made before the programs can be linked to other compilers. Similar modifications are needed for transporting the codes to microcomputers with 80286/80287. A two-byte integer RNG using the "additive congruential" algorithm has also been implemented.

The performance of these two RNG are very good in terms of their speed, long period and consistently producing reliable results in statistical tests described in section 4. For the sake of comparison, we have also tested other RNGs implemented in the most widely used compilers. We found that the real RNG in Turbo-Pascal v3.0 has very bad statistical properties. In all the statistical tests we have done, we found that the two-dimensional plot of consecutive pairs of random numbers in the sequence is the most effective, simple and illustrative test. Other useful tests are the triple distribution chi-square test and the run length test. With the advance of graphic capabilities in microcomputers, we could foresee that the two-dimensional plot of projections of consecutive n -tuples of random numbers in the sequence would

serve as a simple and effective test of the randomness.

References

- [1] K. Binder, Monte Carlo Methods, ed. K. Binder (Springer-Verlag, Berlin, 1979).
- [2] K. Wilson, Phys. Rev. D10 (1974) 2445.
- [3] R.C. Tausworth, Math. Comput. 19 (1965) 201.
- [4] S. Kirkpatrick and E. Stoll, J. Comput. Phys. 40 (1981) 51.
- [5] D.H. Lehmer, Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery (Harvard University Press, Cambridge, 1951) p. 141–146.
- [6] T. Hull and A. Dobell, Soc. Indus. Appl. Math. Rev. 4 (1962) 230.
- [7] IBM Subroutine Library-Mathematics, User's Guide (1971), Program number 5736-XM7; IMSL (1980). The IMSL Library Reference Manual, Edition 8, IMSL, Inc., Houston. IMSL ROUTINE NAME-GGUBS.
- [8] M. Creutz, L. Jacobs and C. Rebbi, Phys. Rep. C95 (1983) 201.
- [9] C. Kalle and S. Wansleben, Comput. Phys. Commun. 33 (1984) 343.
- [10] S.W. Golomb, Sequences with Random Properties (Martin Co., Baltimore, MD, 1955).
- [11] N. Zierler, J. Soc. Indust. Appl. Math. 7 (1959) 31.
- [12] T.G. Lewis and W.H. Payne, J. Assoc. Comput. Mach. 20 (1973) 456.
- [13] N. Zierler, Inf. Control 15 (1969) 67.
- [14] N. Zierler and J. Brillhart, Inf. Control 13 (1968) 541, 14 (1969) 566.
- [15] D. Knuth, The Art of Computer Programming, vol. 2, 2nd edition (Addison-Wesley, Reading, MA, 1981).

Table 6

Run Length Statistics (20 samplings with $N = 10^5$ each. Mean values and standard deviations are shown)

	Total runs	$n(1)$	$n(2)$	$n(3)$	$n(4)$	$n(5)$	$n(6)$	$n(7)$	$n(8)$	$n(9)$
Theoretical expectation	66667	41667	18333	5278	1151	203	30	4	0	0
TRCG	66697 ± 117	41717 ± 191	18335 ± 113	5245 ± 47	1160 ± 32	204 ± 14	30 ± 5	4 ± 2	1 ± 1	0 ± 0
ITRCG	66630 ± 106	41649 ± 188	18342 ± 97	5265 ± 53	1135 ± 31	205 ± 13	29 ± 4	4 ± 2	0 ± 1	0 ± 0
TURBO-PASCAL real	66478 ± 138	41414 ± 233	18317 ± 135	5331 ± 48	1168 ± 26	211 ± 19	33 ± 6	4 ± 2	1 ± 1	0 ± 0
TURBO-PASCAL integer	66682 ± 124	41751 ± 175	18290 ± 92	5264 ± 82	1146 ± 33	197 ± 18	30 ± 4	4 ± 2	1 ± 1	0 ± 0
MS-BASIC COMPILER real	66635 ± 99	41638 ± 146	18302 ± 78	5302 ± 50	1157 ± 35	202 ± 12	30 ± 5	4 ± 2	1 ± 1	0 ± 0

- [14] MS-BASIC Compiler v5.0, User's Manual, Microsoft Inc. (1982).
- [15] TURBO-PASCAL v3.0, User's Manual, Borland International (1985).
- [16] M. Abramowitz and A. Stegun, Handbook of Mathematical Functions (Dover, New York, 1972).
- [17] P.A.W. Lewis, A.S. Goodman and J.M. Miller, IBM Systems J. 8 (1969) 136.