Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

SOME EFFICIENT RANDOM NUMBER GENERATORS FOR MICRO-COMPUTERS

Arne Thesen, Zhanshan Sun and Tzyh-Jong Wang

Department of Industrial Engineering,
University of Wisconsin-Madison, Madison, WI 53706

ABSTRACT

The relatively slow speed and small word size of the current crop of micro-computers cause the efficient production of pseudo-random numbers on these machines to be considerably more difficult than on larger computers. As a consequence, some micro-computer-based algorithms are excessively time comsuming, while other algorithms trade off speed against "randomness". To alleviate this problem, we present in this paper several families of pseudo random number generators explicitly designed for use on micro-computers. Some of these are adaptations of well known generators to the micro-computer environment, others are new or lesser known algorithms designed to overcome some of the restrictions intrinsic to the micro-computer's 8 and 16 bits environments. For each generator the basic algorithm is discussed and FORTRAN and/or Pascal implementations for IBM-PC 's with and without the 8087 co-processor are presented. Values of coefficients leading to pseudo random number streams with good statistical properties are recommended and an empirical evaluation of computational efficiency is offered.

INTRODUCTION

The widespread availabiltiy of micro-computers has encouraged the implementation of micro-computer based software that previously was only available on larger computers. Frequently this software is adapted from earlier designs for larger computers. In some cases clever design and a heavy use of program and data overlays are all that is needed to "shoe-horn" these large software packages onto smaller computers. In other cases, such as for the class of algorithms discussed here, a fundamental redesign of the underlying algorithms may be required.

The most obvious reason for the failure of pseudo-random number generators designed for larger computers to work properly on micro-computers is the difference in word size (down from 32 bits to 16 bits). However merely adjusting the algorithm to reflect the reduced word size does not solve the problem. This is because:

1) The limited word size (16 bits) seriously limits the number of unique integers that can be produced using conventional congruential generators.

2) The relative speed of different arithmetic operations is not the same on micro-computers as on larger computers.

3) Real numbers are usually represented with greater precision than integers (4 to 8 bytes vs. 2 bytes).

During the research leading to this paper, we investigated the performance of thousands of different combinations of procedures and/or coefficients for generation of random numbers on the IBM-PC and similar machines. Most looked promising, but failed to perform well when subjected to the tests discussed later in this paper. A few performed extremely well on our tests. Representative implementations of these algorithms in FORTRAN-77 and Pascal are presented in the text.

DESIGN CONSIDERATIONS

Users of pseudo-random number generators are concerned with the "randomness" of the numbers generated as well as with the programming and computational effort required to implement it. Many trade-offs are therefore made when a specific generator is chosen for a given application. In this section we will briefly discuss some of these.

Fundamental Properties

The sequence of numbers generated by a pseudo-random number generator is not a random sequence, rather, it is a repeating fixed sequence of numbers that pass many reasonable empirical tests for "randomness". For example, a simple linear congruential generator may generate the following repeating permutation of the integers between 0 and 15:

1,14,7,12,13,10,3,8,9,6,15,4,5,2,11,0,1....

The length of the repeating sequence is called the **period** of the generator, and the (ordered) set of numbers generated in a period is referred to as a **cycle.** The **resolution** of a generator is the smallest possible difference between two unequal numbers produced by the generator.

It is desirable to have a generator with as long a period as possible. In addition, it is desirable to have a generator where a cycle contains multiple occurrences of any one integer. (Without this feature, the interval between like numbers in the stream will be equal to the cycle size for all numbers in the stream.) Finally a generator of random integers should have a resolution of one.

Computer language requirements

A competent computer programmer is able to implement any pseudo-random number generator in almost any higher level computer language. Even so, higher level languages impose several restrictions on the implementation of pseudo-random number generators. Among the more common problems are:

1. The value of local variables (such as seeds) might be lost between calls to a procedure.
2. Access to individual bits or bytes of a variable might be difficult.
3. Special purpose arithmetic or logical operations (such as MOD or XOR) might not be available.
4. Parameter passing is frequently time consuming.

In this paper, we illustrate how these difficulties can be overcome.

Statistical Properties

Even though the streams of numbers generated by pseudo-random number generators are repeatable and therefore are **not** random, our intention is to use these numbers in lieu of truly random numbers. The generated streams must therefore exhibit the same behavior as truly random numbers in the application of interest.

The concept of "randomness" embodies many different statistical properties, and a single statistical test is not sufficient. Instead, different statistical tests are required for different properties. The properties tested for in the research reported here are:

1. Uniformity of distribution.
2. Randomness of sequence
3. Absence of autocorrelation.

The specific tests used for each property are discussed in [6]. Needless to say, all the procedures and coefficients presented in this paper yields pseudo-random number streams that pass these tests.

INTEGER (2 BYTE) GENERATORS

The Linear Congruential Generators

The linear congruential (LC) algorithm is perhaps the best known and most widely used procedure for computer generation of pseudo-random integers. This is not surprising as the algorithm is both easy to understand and easy to code in almost any programming language. Furthermore , the behavior of LC generators for 32-bit machines has been extensively studied and many good versions are available in the open literature.

For 16-bit micro-computers, we are not so lucky. The reduced word size imposes intrinsic limits on "randomness" that are much less attractive than those present when larger word sizes are used. It is therefore essential that designers and users of 16-bit LC algorithms fully understand how these algorithms work and what their restrictions are. Designers need this knowledge to optimize the performance of their algorithms (apparently trivial changes can quadruple the count of unique numbers generated), and users need this knowledge to decide if an alternative algorithm should be used.

Algorithm

The Linear Congruential algorithm produces a new random number from the previous one through the congruential relationship shown in Algorithm 1:

```
        I[i+1]  = ( a * I[i] + c ) MOD m

where I[i]    = the i-th number produced by the
                generator
      I[i+1]  = the (i+1)-th number produced by
                the generator
      a,c,m   = constants
```

Algorithm 1: Linear congruential generator.

Using this relationship, I[i+1] is computed as the remainder of (a * I[i] + c) divided by m. For example, if a = 13, c = 1, m = 16 and I[3] = 7, then I[4] is computed as the remainder of (13*7+1)/16 or I[4] = 12.

An important feature of LC generators is the fact that each period contains **at most** one occurrence of each of the integers in the range 0 .. (**m** - 1). Therefore, the interval between any two like integers is fixed, and the maximum period is **m**.

The properties of LC generators have been extensively studied. Knuth [2] provides guidelines for selecting the values for the coefficients **a, c** and **m** and recommends specific values for large computers. Recommended values for micro-computers are presented later in this section.

Implementation

To minimize computational effort, **m** is chosen to be the largest integer that the computer can represent + 1 (2**15 or 32768 on most micro-computer systems). This has the advantage that the MOD operation in the Linear Congruential relationship is performed automatically when the term (a * I[i] + c) is formed through integer overflow.

To minimize the importance of the initial seed, we should generate **all** non-negative integers less than **m**. It can be shown (Knuth [2]) that this is always possible if c and a are chosen according to certain rules. However, note that no guarantees are made regarding the statistical properties of the resulting number sequence. One family of generators satisfying these rules is:

```
      I[i+1] = ( a * I[i] + 1 ) MOD m

where a  =  k * 4 + 1  (k= 0,1,....).
```

A Pascal implementation of this generator is given in Program 1 and a FORTRAN implementation is given in Program 2:

```
{------------------------------------------}
 Function I_lcg(var seed: integer): integer;
{------------------------------------------}
const
  mult =  3993;
begin
  seed := seed * mult +1;
  if seed < 0 then
    seed := seed + maxint +1;
  I_lcg := seed;
end;
```

Program 1: Two byte congruential generator in Pascal

```
C--------------------------
      FUNCTION ILCG(ISEED)
C--------------------------
      INTEGER*2 ISEED
      DATA MULT/3993/,MAXINT/32767/
      ISEED = ISEED * MULT + 1
      IF(ISEED.LT.0) ISEED = ISEED + MAXINT + 1
      ILCG = ISEED
      RETURN
      END
```

Program 2: Two byte congruential generator in FORTRAN

Both implementations rely on integer overflow for their MOD operation. While this is efficient, it introduces two problems:

- Integer overflow may cause the sign bit to change.
- Some systems stop execution when an overflow condition is detected.

Programs 1 and 2 solve the first problem by resetting the sign bit if it has been changed. Note that this is done by adding 2**15 (=32,767 +1) not by multiplying by minus one. (The latter will cause all the bits to change as two´s complement is used to store negative numbers.)

If the second problem is encountered, it may be solved by including compiler switches in the source code to disable error checking. We have found some Pascal installations where it is impossible to disable the "abort on overflow" feature. In these cases it usually works to define the seed as a global variable of the type [0 .. 65535] and to reset its leading bit if the value of the seed exeeded 32767.

The Multiplier

A total of 8192 different values for a satisfy the condition a = 4*k+1 can be chosen for Algorithm 1. Some of these yield "good" sequences, while others yield "bad" sequences. For example, the following sequence is generated if a is one:

0-1-2-3-4-5-6-7-8-9-10-11-...-32766-32767-0-1-2...

This sequence clearly has a period of m. However, it will not pass any reasonable test for randomness.

As we are not aware of any method for predicting in advance which values of a will result in "good" sequences and which values will result in "bad" sequences, we conducted empirical tests on the output produced by all of the 8192 possible values of a. Some of the values of a that were found to yield statisically "good" sequences are listed in Table 1.

589	1813	2125	2633	3993
4773	5225	5737	5995	6061
7149	11097	11245	12217	20377

Table 1: Good multipliers for 16-bit LC generators.

Remarks

Some readers may feel that a certain degree of simplicity and computational efficiency would be gained when c is set equal to zero. However, we should point out that such generators have only one fourth of the period of the recommended mixed generator. As a consequence, the generator produces two mutually exclusive sets of odd random integers. The actual set produced in any one run depends on the initial seed chosen for that run. Furthermore, the elimination of c has no significant impact on the speed of the resulting procedure.

A Tausworthe generator

Tausworthe[5] has suggested a different approach to the generation of pseudo-random integers. This procedure, which operates directly on bits to form a stream of random bits, has been shown to produce random number sequences that (1) have improved statistical properties over LC generators, and (2) have an arbitrarly long period independent of the word size of the computer used.

Tausworthe generators are not in widespread use. This could be because they are difficult to implement efficently in a higher level language, or because their improved statistical properties are of marginal utility on larger computers. On micro-computers, the situation is quite different. Here the improvement in period length and in statistical properties is quite substantial, and, well written Tausworthe generators are not necessarely more time consuming than other classes of generators.

Algorithm

The basic procedure of a Tausworthe type generator is illustrated in Figure 1:

```
       B[i-p]  B[i-r]    B[i]
  B = . . x . . . . . x . . . . . x . . . . .
          |           |           ^
          +-->xor<--+           |
                |                 |
                +---------------+
```

Figure 1: Relationship between bits in a Tausworthe generated bit stream.

Here B is defined as a sequence of bits, and the relationship between individual bits in the sequence is defined in Algorithm 2:

```
B[i] = B[i-r]  XOR  B[i-p]

where:   i    = any integer,
       r , p  = fixed integers with 0<r<p.
        XOR   =the exclusive OR operator
               yielding 0 if the terms are
               equal and 1 if they are not
```

Algorithm 2: Tausworthe generator

When r and p are properly selected (as primitive trimodals [8]), the maximum period of the stream (B) is 2**p - 1.

Implementation

In Program 3 we present a FORTRAN implementation of the Tausworthe based on an idea first proposed by Lewis and Payne [3]. To avoid the need to access individual bits, the algorithm maintains 15 independent and parallel streams of bits, and the exclusive OR operation is performed on all bits at once. Since each of the independent bit streams have a period of 2**p - 1, the resulting stream of integers will also have a period of 2**p - 1.

```
C--------------------------------------------
      INTEGER*2 FUNCTION IXOR(FIRST,SECOND)
C--------------------------------------------
        INTEGER*2 FIRST,SECOND,F1,S1,F2,S2,
     *            X,Y
        LOGICAL*2 FF1,SS1,FF2,SS2,XX,YY
        CHARACTER*1 CF1(2),CS1(2),CF2(2),
     *            CS2(2),CX(2),CY(2)
        EQUIVALENCE (F1,FF1),(S1,SS1),
     *            (F2,FF2),(S2,SS2),
     *            (X,XX),(Y,YY)
        EQUIVALENCE (F1,CF1(1)),(S1,CS1(1)),
     *            (F2,CF2(1)),(S2,CS2(1)),
     *            (X,CX(1)),(Y,CY(1))
        DATA MAXINT/65535/
        F1=FIRST
        S1=SECOND
        F2=MAXINT-F1
        S2=MAXINT-S1
        XX=FF2.AND.SS1.OR.FF1.AND.SS2
        CF1(1)=CF1(2)
        CF2(1)=CF2(2)
        CS1(1)=CS1(2)
        CS2(1)=CS2(2)
        YY=FF2.AND.SS1.OR.FF1.AND.SS2
        CX(2)=CY(1)
        IXOR=X
        RETURN
        END

C-------------------------
      BLOCK DATA INTTABLE
C-------------------------
        COMMON/TABLE/ITABLE(98)
        DATA ITABLE/
     *    12367,23891,31506,18710,195,22840,
     *    8267,18890,30239,24237,12578,604,
     *    10782,30128,25410,7,18271,21141,
     *    12085,25438,2395,8854,23562,32544,
     *    5796,10976,14721,24781,9690,31984,
     *    23409,3957,14721383,17222,5234,
     *    18963,29006,18273,9815,31802,
     *    19161,28282,14975,25973,32605,141,
     *    30030,7767,29294,9884,19885,27311,
     *    4209,19675,9596,1052,23999,9052,
     *    13660,31035,6578,28125,18883,
     *    10482,5735,13025,24226,32043,82,
     *    20418,13570,32554,99,12326,30454,
     *    19576,15552,20577,12124,26038,
     *    4142,32092,11825,5482,26736,23403,
     *    31196,2762,14193,22213,10746,
     *    24414,31884,11266,579,29011,2262/
        END
C--------------------------
      FUNCTION ITAU(IFIRST)
C--------------------------
        INTEGER*2 IFIRST,S,M,N
        COMMON/TABLE/ITABLE(98)
        IF (IFIRST.LT.0) IFIRST=0
        IFIRST=IFIRST+1
        IF (IFIRST.GT.98) IFIRST=1
        S=IFIRST+27
        IF (S.GT.98) S=1
        M=ITABLE(IFIRST)
        N=ITABLE(S)
        ITAU=ITABLE(IFIRST)
        ITABLE(IFIRST)=IXOR(M,N)
        RETURN
        END
```

Program 3: FORTRAN implementation of a Tausworthe generator of pseudo-random integers.

Three program segments are included in Program 3. The function ITAU(IFIRST) is the actual generator implementing Algorithm 2. Its parameter "IFIRST" is subscript of the array of bits (ITABLE) maintained by the procedure. Its value should not be initialized or changed by the user. The block data segment initializes the same array. Again no changes should be made by the user. Finally the function IXOR is our FORTRAN implementation of a bitwise **xor** operator. The speed of the procedure is improved by approximately 50% if when this function is recoded into assembly language. However the resulting code would not be very portable.

Shuffle Generators

Shuffle generators combine two or more independent generators to produce a single random number stream with (hopefully) improved statistical properties. Of course, this improvement (if any) comes at the cost of reduced computational efficiency. Many elaborate shuffling algorithms have been proposed. The key to success for any of these is the requirement that the driving generators must have periods that are relative prime.

The shuffle algorithm presented here is adapted from Knuth [2]. As shown in Algorithm 3, an internal table of pseudo-random numbers is maintained. Whenever a random number is requested, a random index is generated, and the random number stored at the corresponding location in the table is returned. This entry is then replaced by a new number using the other random number generator.

```
Step 0 : Initialize

        for index := 1 to MaxIndex do
          Table[index] := Irandom1(0,MaxInt);

Step 1 : Draw a random subscript

        index   := Irandom2(1,MaxIndex)
```

```
Step 2 : Obtain value stored at this location

         Irandom := Table[index]

Step 3 : Replace this value

         Table[index] := Irandom1(0,MaxInt)

where :  index    = a random subscript
         Irandom  = the resulting random integer
         Irandom1 = generates random integers
         Irandom2 = generates random integers
         MaxIndex = the size of Table
         Table    = an array of random integers
```

Algorithm 3: Knuth´s shuffle algorithm

The resulting procedure (Program 4) produces random number streams of quite (Knuth says exceptionally) long periods.

```
C-----------------------
      FUNCTION ISELECT  (I)
C-----------------------
        INTEGER*2 ISEED,I
        CHARACTER*1 BYTE(2)
        EQUIVALENCE (ISEED,BYTE(1))
        DATA MULT/6061/,IPRIME/32749/,MAXINT/32767/
        ISEED=I
 1000   ISEED=ISEED*MULT+1
C       B(2) IS NOW IN RANGE 0..255
        IF (ISEED.LT.0) ISEED=ISEED + MAXINT + 1
C       B(2) IS NOW IN RANGE 0..127
C
C       FORCE DIFFERENT PERIOD
C
        IF (ISEED.GE.IPRIME) GOTO 1000

        ISELECT=ICHAR(BYTE(2))
        I=ISEED
        RETURN
        END

C----------------------------------------
      FUNCTION ISHUF(ISEED1,ISEED2,INIT)
C----------------------------------------
        COMMON/TABLE/ITABLE(128)
        INTEGER*2 ITABLE,ISEED1,ISEED2,INIT,INDEX
        IF (INIT.EQ.1) THEN
           INDEX=ISELECT(ISEED2) + 1
           ISHUF=ITABLE(INDEX)
           ITABLE(INDEX)=ILCG(ISEED1)
           RETURN
        ENDIF
        DO 1010 INDEX=1,128
 1010      ITABLE(INDEX)=ILCG(ISEED1)
        INIT=1
        RETURN
        END
```

Program 4: Two byte shuffle generator. INIT must have a value of zero when ISHUF is called for the first time.

The two driving generators used in this procedure are ILCG (Program 2) and ISELECT (Program 4). ILCG determines the actual values produced by the generators, and ISELECT determines the order in which they are returned. ISELECT is an example of a random byte generator. Note that ISELECT has an extra step inserted to force a period other than the default of 32768. Random byte generators are further discussed in the following section.

SINGLE BYTE GENERATORS

The byte is the basic information building block in a micro-computer. Two adjacent bytes are used to represent an integer and four to eight adjacent bytes are used to represent a floating point number. An application of a random byte generator was shown in the previous section where such a generator was used to produce subscripts in the range 0 .. 127. In a later section, we will show how to construct random deviates of more complex types from a stream of random bytes.

A Congruential Generator

While is is easy to develop an LC generator operating on a single byte, the short period of the resulting stream of numbers causes such a generator to have limited value.

For satisfactory results we will therefore produce random bytes from a stream of LC generated integers. It is tempting to develop a procedure that first uses the higher order and then the lower order byte of the integer. **This is extremely dangerous** as the guarantee of randomness for an integer does not extend to the bytes that make up that integer. In particular the lower order byte of an integer produced by a LC generator is likely to exhibit extremely poor statistical properties. However, if the multiplier is carefully chosen, it is possible to obtain a stream of random bytes by returning the high order byte (only) of a random integer). Function ISELECT in Program 4 is a FORTRAN implementation of a byte generator returning the higher order byte of a positive integer. (Note that the resulting byte is in the range [0 .. 127]). A Pascal implementation of a byte generator returning bytes on the range [0 .. 255] is given in Program 5.

```
(* var
    seed : record
             case integer of
               1: (int : integer) ;
               2: (lsbyte : byte  ;
                   msbyte : byte) ;
             end ;              *)
function RByte : byte ;
const
  Multiplier = 3993 ;
begin
  seed.int := Multiplier * seed.int + 1 ;
  RByte := seed.msbyte ;
  if seed.int < 0 then
   seed.int := seed.int + maxint + 1 ;
end ;
```

Program 5: Congruental generator of random bytes

Here the external seed for the congruential generator is of type integer. This seed is immediately copied to an internal variable that is "EQUIVALENCEd" to be both an integer **and** an array of two bytes. The standard congruential operations are performed on this internal variable. Then its most significant byte is captured and the new seed value is copied to the externally defined integer.

A Tausworthe Generator

In Program 3 we showed that a random integer could be developed from fifteen parallel random bit streams using a Tausworthe generator. Since the underlying algorithm was not constrained by the word size of the computer used, we can easily modify it to generate a random byte from eight parallel streams of random bits. Due to the simplicity of this extension, we will not provide a FORTRAN implementation at this time. Instead we show in Program 6 how such an algorithm can be implemented in Pascal. In this implementation we exploit the fact that our Pascal compiler (TurboPascal) provides a built-in **xor** operator. The procedure differs further from Program 3 in that a separate initialization routine is used to generate initial values for the byte table and the offset pointers.

```
(* const
     pminus1 = 97 ; { p = 98 ; q = 27 }

   var
    ByteTable : array[0..97]of byte ; *)

procedure TauInit(var f,s : integer) ;
var
  BitIndex,count : integer ;
  WorkTable : array[0..97]of byte ;
begin
  for count := 0 to 97 do
   begin
     ByteTable[count] := 255 ;
     WorkTable[count] := 0   ;
   end ;
  f := pmunus1 ;
  s := 26 ;
  for BitIndex := 1 to 16 do
   begin
     for count := 1 to 9800 do
      begin
        if f < pminus1 then
          f := f + 1
        else
          f := 0 ;
        if s < pminus1 then
          s := s + 1
        else
          s := 0 ;
        ByteTable[f] :=
               ByteTable[f] xor ByteTable[s] ;
      end ;
     if BitIndex > 8 then
      for count := 0 to 97 do
       begin
         if odd(ByteTable[count]) then
              WorkTable[count] :=
                (WorkTable[count] div 2) + 128
         else
              WorkTable[count] :=
                 WorkTable[count] div 2 ;
         ByteTable[count] :=
              ByteTable[count] div 2 ;
       end ;
   end ;
 for count := 0 to 97 do
    ByteTable[count] := WorkTable[count] ;
  f := pmunis1 ;
  s := 26 ;
 end ;
```

```
function RByte(var f,s : integer) : byte ;
begin
  if f < pminus1 then
    f := f + 1
  else
    f := 0 ;
  if s < pminus1 then
    s := s + 1
  else
    s := 0 ;
  RByte := ByteTable[f] ;
  ByteTable[f] := ByteTable[f] xor ByteTable[s] ;
end ;
```

Program 6: Tausworthe generator of random bytes.

LONG INTEGER (4 BYTE) GENERATORS

Congruential Generators

The LC generator has been proven to give satisfactory results for computers capable of performing 32-bit arithmetic in hardware. This is not yet possible on micros. However, it is possible to emulate 32-bit arithmetic in software. We will review two possible approaches. Unfortunately neither of these yield entirely satisfactory results.

The most appealing approach is to exploit the fact that the standard word size in FORTRAN-77 is four bytes. One would therefore think that our Standard FORTRAN-77 compiler should automatically solve our problem. Not so. This is because long integer overflow unfortunately results in a bit string of ones rather than in the least significant bits of the answer. The situation is even more difficult when Pascal is used, as our Pascal compiler does not even support long integers.

In Program 7 we present an implementation of a four byte LC generator that overcomes these problems.

```
C     ---------------------
      FUNCTION ILONG(ISEED)
C     ---------------------
      INTEGER*4 A,PRIME,ISEED,BIT15,BIT16
      INTEGER*4 XHI,XLO,LLO,FHI,K
      DATA A/16807/,BIT15/32768/,BIT16/65536/
      DATA PRIME/2147483647/
      XHI = ISEED / BIT16
      XLO = (ISEED-XHI*BIT16) * A
      LLO = XLO / BIT16
      FHI = XHI * A + LLO
      K = FHI / BIT15
      ISEED = (((XLO-LLO*BIT16)-PRIME)+
     1           (FHI-K*BIT15)*BIT16)+K
      IF(ISEED.LT.0) ISEED = ISEED + PRIME
      ILONG = ISEED
      RETURN
      END
```

Program 7: Congruential generator of 4 byte integers

Due to Schrage [4], program 7 avoids integer overflow by (1) converting the four byte seed to two 2 byte integers, (2) performing the congruential operations on these integers, and (3) combining these to form a new long integer. An added feature of Program 7 is the fact that it takes the modulus of the largest prime the computer can represent rather that of the largest number it can represent.

This has the potential of improving the statistical quality of the resulting random number stream. Our evaluation of Program 7 confirms the fact that it yields integers with good statistical properties.

FLOATING POINT GENERATORS

Conventional procedures

Real valued random deviates are readily obtained from integer deviates by mode conversion (from integer to real) and by scaling (from 0.0-32767.0 to 0.0-1.0). One such procedure is illustrated in Program 8.

```
C-------------------------
      FUNCTION UNIF(ISEED)
C-------------------------
      INTEGER*2 ISEED
      DATA MULT/2125/,MAXINT/32767/
      ISEED = ISEED * MULT + 1
      IF (ISEED.LT.0) ISEED = ISEED + MAXINT + 1
      UNIF = ISEED * 3.051851E-5
      RETURN
      END
```

Program 8: Congruential generator for variables on [0.0 - 1.0]

Note that the floating point number is produced from the integer by **multiplying** by a constant equal to 1/32767. This is a substantially faster uperation than the more obvious step of **dividing** by 32767.

Two serious problems restrict this program's usefulness for micro-computers:

1) Floating point arithmetic is particularly slow on most micro-computers. Program 8 is about 4 times slower than its integer counterpart.

2) Most micro-computer languages use four bytes to store a floating point number while they use only two bytes to store an integer. Program 8 has inherited the cycle and period restrictions of our two byte integer generators.

In the following section, we present a different approach to the generation of floating point random numbers that overcomes these problems. This procedure is discussed in more detail in [7].

A Construction Algorithm for Floating Point Deviates

A random variable u on [0-1) can be expressed as a function of a random exponent e and a random mantissa m as follows:

$$u = \frac{m}{M} * 2^{e}$$

Where u = random variable on [0-1)
- e = random variable drawn from the geometric distribution with mean 0.5.
- M = a constant
- m = Uniformly distributed random variable on [M/2,M).

An procedure exploiting this notational convention to genetate numbers on [0.0 - 1.0) is presented in Algoritm 4:

```
 I. INITIAL ASSIGNMENTS

      e = 0             (Correct value if u > 1/2)
      m = U(0,M)        (Uniform deviate on 0-M)

 II. IS ADDITIONAL WORK REQUIRED?

      If m >= M/2       (this happens half the time)
      then go to step IV
      else m = m + M/2 (mantissa must be >  M/2)

III. ADJUST EXPONENT

      A. Draw random byte(s)
               until a nonzero byte is found:

            k = RandomByte
            while k = 0 (this happenswith a
                           probability of 1/256)
              e = e - 8
              k = RandomByte

      B. Scan the byte for the first nonzero bit:

            while k < 128
              k = k * 2
              e = e - 1.

 IV. RANDOM VARIABLE IS u = f(e,m)
```

Algorithm 4: Building u from e and m

The algorithm starts out by assigning a value of zero to the exponent and a random value to the mantissa (defining a number on [0.5 .. 1.0)). A check is made to see if the resulting mantissa has a valid value (there is a 50% probability that this is true). If so, the algorithm stops as a valid number has been produced.

If the mantissa is not valid (i.e. it is less than M/2), then M/2 is added to m to make it valid, and a procedure for generating the random exponent required for numbers in the range [0.0 .. 0.5) is entered. This procedure exploits the fact that the number of bits preceding the first bit with a value of one in a stream of random bits follows the geometric distribution with p = 0.5. Random bytes are drawn until a nonzero byte is found. The resulting exponent is computed as the negative value of eight times the number of zero valued bytes plus the number of consective zero valued bits in the last byte.

Implementation

Our implementation of algorithm 4 is based on the IEEE standard[1] for representation of floating point numbers shown in Figure 2:

```
    bit:  31  30   ...   23 22        ...          0
         +----+-------------+---------------------+
content: |Sign|Biased expon.|mantissa less 1st bit |
         +----+-------------+---------------------+
```

Figure 2: IEEE Standard for floating point representation. Note that the leading bit of the mantissa is omitted as it is always a one.

For numbers between zero and one, the sign bit is zero. The biased exponent is 126 if the number is between 1 and 1/2; it is 125 if the number is between 1/2 and 1/4 etc. Bits 0 .. 22 represents the 23 least significant bits of the 24-bit mantissa. For a random variable between 0 and 1 , these bits have an equal likelihood of being zeros or ones. The most significant bit of the mantissa is omitted as it always has a value of one.

Our implementation of a FORTRAN algorithm exploiting this data structure is shown in Program 9. This subroutine follows Algorithm 4 closely. The only difficulty in implementation arises from the fact that the boundary between the eight bit exponent and the mantissa is not on a whole byte boundary. This problem is solved by treating those seven bits of the exponent that resides in byte(4) separately from the bit that forms the most significant bit of byte (3). It should be clear from the program annotation how this solves the problem. In Program 10 we give the equivalent Pascal implementation of Algorithm 4.

```
C----------------------------------
      FUNCTION UNIF(ISEED,JSEED)
C----------------------------------
        INTEGER*2 ISEED,JSEED,IEXPO
        CHARACTER*1 B(4),RBYTE
        EQUIVALENCE (X,B(1))

C  FILL MANTISSA, SET  BITS 2-8 OF EXPONENT

        CALL RANDOM(ISEED,B(1))
        CALL RANDOM(ISEED,B(2))
        CALL RANDOM(ISEED,B(3))
        IEXPO = 63

C  EXPONENT BYTE IS NOW 127 OR 126, QUIT IF 126

        IF (B(3).LT.128) GOTO 1030

C  LEAST SIGNIFICANT BIT OF EXPONENT IS A ONE
C  FIND NON ZERO RANDOM BYTE

        IEXPO = 66
 1010   CALL RANDOM(JSEED,RBYTE)
        IEXPO = IEXPO - 4
        IF (RBYTE.EQ.0) GOTO 1010

C EXPONENT BYTE IS IN [125, 117, 109, ...]

C SCAN RBYTE FOR FIRST NONZERO BIT

        IF (RBYTE.GT.127) GOTO 1030
        IF (RBYTE.GT.63) GOTO 1020

C   SUBTRACT 2 FROM EXPONENT BYTE
        IEXPO = IEXPO - 1
        IF (RBYTE.GT.31) GOTO 1030
        IF (RBYTE.GT.15) GOTO 1020

C   SUBTRACT 2 FROM EXPONENT BYTE
        IEXPO = IEXPO - 1
        IF (RBYTE.GT.7) GOTO 1030
        IF (RBYTE.GT.3) GOTO 1020

C   SUBTRACT 2 FROM EXPONENT BYTE
        IEXPO = IEXPO - 1
        IF (RBYTE.GT.1) GOTO 1030

C LEAST SIGNIFICANT BIT OF EXPONENT IS ZERO
 1020   B(3) = CHAR(ICHAR(B(3)) - 128)
 1030   B(4) = CHAR(IEXPO)
        UNIF = X
        RETURN
        END
```

Program 9: FORTRAN-77 implementation of the generator

Discussion

While a fixed number of random bytes are always used to generate a random mantissa, the number of bytes **n** used to generate a random exponent is itself a random variable. It can be shown that the distribution of **n** is:

$$p(n=i) = \{ 1/2 \qquad \text{for } i = 0$$

$$p(n=i) = \{ 255 * (1/256)^i / 2 \qquad \text{for } i = 1,2,3,\ldots.$$

and the expected value **n** is 128/255 = 0.5019607.

Separate random number streams must be used for the exponent and the matissa. Programs 9 and 10 give poor results when a single stream is used for both.

```
(* var     { copy this global declaration as is!!}
   u : record case integer of
           1: (unif : real) ;
           2: (m3 : byte;
               m2 : byte;
               ml : byte;
               exponent  : byte);
        end ;                       *)
function uniform : real ;
var
  k : byte ;
begin
  with u do begin
    m3 := RBytel ;
    m2 := RBytel ;
    ml := RBytel ;
    exponent := 63 ;
    if ml >= 128 then begin
      exponent := exponent - 1 ;
      k := RByte2 ;
      while k = 0 do begin
        exponent := exponent - 4 ;
        k := RByte2 ;
      end ;
      if k < 128 then
        if k >= 64 then
          ml := ml + 128
        else
          if k >= 32 then
            exponent := exponent - 1
          else
            if k >= 16 then begin
              ml := ml + 128 ;
              exponent := exponent - 1 ;
            end
            else
              if k >= 8 then
                exponent := exponent - 2
              else
                if k >= 4 then begin
                  ml := ml + 128 ;
                  exponent := exponent - 2 ;
                end
                else
                  if k >= 2 then
                    exponent := exponent - 3
                else begin
                  ml := ml + 128 ;
                  exponent := exponent - 3 ;
                end ;
    end ;
    uniform := unif ;
  end ;
end ;
```

Program 10: MS-Pascal implementation Algorithm 4.

EVALUATION

All the generators presented here were subjected to extensive performance tests. This included statistical tests for distribution, sequence and autocorrelation. All of the generators presented here passed all of these tests. The reader is refered to [6] and [7] for further information.

The generators were also subjected to extensive timing tests. These tests were performed by measuring the time required to generate 32,000 random numbers on two IBM-PC's one with and one without the 8087 co-processor. The results of these are summarized in Tables 2 and 3.

The reader is warned against reading too much into minor differences in execution times. Such differences are as likely to be caused by differences in programming styles and data transfer methods as by intrinsic algorithmic performance differences. For example we found that the Tausworthe byte generator (Program 6) performed three to four seconds faster in our tests when the resulting byte was maintained as a global variable rather than returned through the function.

A somewhat unexpected result was the fact that the 8087 numeric co-processor does not significantly improve the speed of any of the integer generators. We have verified the fact that this is because our compliers do not use the powers of the 8087 to perform integer arithmetic.

	Congruential		Tausworthe		Shuffle	
	FORTRAN	Pascal	FORTRAN	Pascal	FORTRAN	Pascal
Byte	7	6	15	8	19	24
I*2	7	8	21	8	17	24
I*4	61	na	29	na	24	na

Table 2: Time to generate 32,000 integers on IBM-PCs using different algorithms. Note that the 8087 co-processor does not improve execution speed.

Table 2 summarizes the run times for our integer generators. We note that the simple congruential generators were consistently faster than the other generators. Users not concerned with the short period of these generators will therefore do well with Programs 1 or 2 as their main source of pseudo-random integers. Pascal users concerned with the short period of these generators may want to use a Tausworthe generator (Program 6) while FORTRAN users may want to use a shuffle algorithm (Program 4).

	8088		8087	
	FORTRAN	Pascal	FORTRAN	Pascal
Congruential	48	102	10	17
Build	26	17	26	17
Shuffle	59	119	22	38

Table 3: Time to generate 32,000 random floating point numbers using different algorithms. Note that the 8087 substantially improves execution speed.

Table 3 summarizes the run times for the floating point generators. We now observe a significant improvement in the performance of most algorithms when the 8087 is used to perform floating point arithmetic. However, algorithm 4 which does not use the 8087 was found to be the fastest procedure for both FORTRAN and Pascal implementations if the 8087 is not available. This is exciting, as the resulting random number stream has substantially better statistical properties than the conventional LC generator. FORTRAN users with access to a machine with an 8087 will save some time if they use the LC generator (Program 8). Pascal users will not gain speed by making this switch.

Conclusion

This paper presents the finding of an extensive evaluation of thousands of different combinations of algorithms and constants for micro-computer based random number generators. All of the generators presented here have been shown to pass reasonable tests for uniformity of distribution, randomness of sequence and absence of autocorrelation. In addition each generator dominates the others in at least one of the dimensions of speed, period and portability.

We had expected to observe a tradeoff between speed and "randomness", however, no such relationship was observed. In fact the fastest generator of floating point numbers also exhibited the longest period. We also had expected to observe a strong relationship between programming style and computational efficiency. This expectation was confirmed. Clever use of nonstandard language features improves speed. Likewise, the manner in which data is transferred to and from the procedures has a significant (15% up to 30%) effect on relative performance. Finally we were surprised to learn that the 8087 co-processor does not improve the efficiency of our integer generator.

However, we feel that the most important lesson learned from this study is the fact that we were completely unable to predict in advance whether or not a given algorithm would exhibit good or bad statistical properties. The likelihood of improving an algorithm through ad hoc changes are slim at best.

BIBLIOGRAPHY

1. Intel Corp., **8232 Arithmetic Processing Unit,** Preliminary Data Sheet, 1981.

2. Knuth, **The Art of Computer Programming.** Addison-Wesley, 1969.

3. Lewis, T.G. and Payne W.H., **Generalized Feedback Register Pseudorandom Number Generator.** JACM vol. 20, no. 3, July 1973.

4. Schrage,L., **A More Portable Fortran Random Number Generator**, ACM Transactions on Mathematical Software, pp 132-138, 1979.

5. Tausworthe, R.C., **Random Numbers Generated by Linear Recurrence, Modulo Two,** Math. Comput. 19 (1965) 201-209.

6. Thesen, Arne, **An Efficient Generator of Uniformly Distributed Random Deviates Between Zero and One.** Technical Report. Mathematics Research Center, University of Wisconsin-Madison, 1983. To appear in Simulation.

7. Thesen, Arne and Tzyh-Jong Wang, **Some Efficient Random Number Generators for Micro Computers** MRC Technical Summary Report #2562 Mathematics Research Center, University of Wisconsin- Madison, 1983.

8. Zierler, Niel and John Brillhart, **On Primitive Trinomials (Mod 2),** Information and Control, Vol. 13 pp 541-554, 1968.