Fast pseudorandom-number generators with modulus 2^k or $2^k - 1$ using fused multiply–add

by R. C. Agarwal
R. F. Enenkel
F. G. Gustavson
A. Kothari
M. Zubair

Many numerically intensive computations done in a scientific computing environment require uniformly distributed pseudorandom numbers in the range (0, 1) and (−1, 1). For multiplicative congruential generators with modulus 2^k, $k \leq 52$, and period 2^{k-2}, we show that the cost per random number for these two distributions is 3 and 3.125 multiply–adds on RS/6000® processors. Our code, on the IBM POWER2 Model 590, produces more than 40 million uniformly distributed pseudorandom numbers per second for both ranges (0, 1) and (−1, 1). Additionally, our code sustains the 40 million per second rate for data out of cache. The Numerical Aerodynamic Simulation (NAS) parallel benchmarks use a linear congruential generator with modulus 2^{46}. Our result is about 50 times faster than the generic implementation given in the benchmarks. The extra-accuracy fused multiply–add instruction of RS/6000 machines combined with a few algorithmic innovations gives rise to the 50-fold increase. If IEEE 64-bit arithmetic is used with our Fortran code on POWER and PowerPC® architectures, the results we obtain are bit-wise identical to the generic algorithms. The paper gives several illustrations of a general technique called the Algorithm and Architecture approach. We demonstrate herein that programmer-controlled unrolling of loops is equivalent to "customized vectorization of RISC-type code." Customized vectorization is more powerful than ordinary vectorization, and it is only possible on RISC-type machines. We illustrate its use to show that RS/6000 processors can compute the distribution (−1, 1) at the rate of 3.125 multiply–adds. We also specify a linear congruential generator that is related to the multiplicative congruential generator referred to above. It has a full period of 2^k, where 2^k is the modulus. The cost per random number

0018-8646/01/$5.00 © 2001 IBM

[in the range (0, 1)] for this generator is four multiply–adds on RS/6000 processors. Our code, on the IBM POWER2 Model 590, for this generator produces more than 30 million uniformly distributed pseudorandom numbers per second for the range (0, 1). We show that this generator is "embarrassingly parallel," or EP. Using the Algorithm and Architecture approach, we describe a new concept called "generalized unrolling." Finally, we present a multiplicative congruential generator for which the modulus is not a power of 2. Such a generator, as well as one with modulus 2^k, is selectable as the generator used in the RANDOM_NUMBER intrinsic function of IBM XL Fortran and XL High Performance Fortran. All of the generators reported here are EP. Using an IBM SP2 machine with 250 wide nodes, it is possible to compute more than ten billion uniform random numbers in a second.

1. Introduction

The extra-accurate fused multiply–add (FMA) operation of the RS/6000* and PowerPC* family of RISC microprocessors offers many opportunities to use mathematical innovation to produce fast algorithms for numerically intensive computation (NIC). In this paper we illustrate this assertion by giving several examples and by demonstrating a 50-fold increase in performance (over generic algorithms [1]) for pseudorandom-number generation. The results obtained are bit-wise identical to the results of the generic algorithms.

For multiplicative congruential generators [2] of the type

$$s_{i+1} = as_i \bmod 2^k,$$

$$x_{i+1} = 2^{-k}s_{i+1} \quad \text{or} \quad 2^{-k+1}s_{i+1} - 1, \tag{1}$$

with $k \leq 52$, we show that the cost per random number for the two distribution intervals (0, 1) and (−1, 1) is respectively 3 and 3.125 multiply–adds on RS/6000 processors.

We also specify a linear congruential generator of the form

$$s_{i+1} = (as_i + c) \bmod 2^k,$$

$$x_{i+1} = 2^{-k}s_{i+1},$$

related to the generator in [2], which has a full period of 2^k. The cost per random number [in the range (0, 1)] for this generator is four multiply–adds on RS/6000 processors.

Finally, we present a multiplicative congruential generator for the interval (0, 1) of the form

$$s_{i+1} = as_i \bmod (2^k - 1),$$

$$x_{i+1} = \frac{s_{i+1}}{2^k - 1}, \tag{2}$$

(discussed in [3]), for which the modulus is not a power of 2. Such a generator, as well as a generator of type (1), is selectable as the generator used in the RANDOM_NUMBER intrinsic function of IBM XL Fortran (XLF) [4] and XL High Performance Fortran (XLHPF) [5]. [By "a generator of type (n)," or "generator (n)," we mean a generator whose description is given by equation(s) (n).]

We first introduce some notation. We use q to represent the modulus, either 2^k or $2^k - 1$, depending on the generator. Arithmetic operators in equations are exact. Operands are IEEE normalized numbers. We use the operators $\oplus$, $\ominus$, $\otimes$, $\oslash$ for IEEE double-precision (64-bit) floating-point arithmetic, and $\mathrm{fl}(x)$ for the correctly rounded double-precision value corresponding to x. In most cases, $x = ab + c$, where a, b, and c are IEEE numbers. Unless otherwise stated, the rounding mode we use is round-to-zero (chop). This leads to the best performance.

Let a, b, and c be arbitrary IEEE 64-bit floating-point numbers. The fused multiply–add operation on RS/6000 and PowerPC computes the correctly rounded $d = \mathrm{fl}(ab + c)$ for any of the four IEEE rounding modes. On RS/6000 machines, all 106 bits of the product $a * b$ are added to c in order to guarantee that d will be correctly rounded for all possible values of a, b, and c. POWER and POWER2 RS/6000 machines can on a continuous basis compute one and two FMAs every machine cycle if the results of the FMAs do not cause any pipeline delays. The pipeline delay for these machines is two or three cycles. Loop unrolling can be used to avoid any pipeline delays for the pseudorandom-number calculation.

We first consider a multiplicative congruential pseudorandom-number generator for which the modulus is a power of 2. See Knuth [2] for a thorough discussion of pseudorandom-number generators. Let $0 < s_0 < 2^k$, s_0 odd, $1 < a < 2^k$, $k \leq 52$, and for $i \geq 0$, let

$$s_{i+1} = as_i \bmod 2^k,$$

$$x_{i+1} = 2^{-k}s_{i+1}. \tag{3}$$

For $k = 46$ and $a = 5^{13}$ we have a specific instance of such a generator. This generator has period 2^{k-2}, and it is extensively used by the Numerical Aerodynamic Simulation (NAS) suite of paper and pencil benchmarks [1]. We mention here that the random-number generator (3) is embarrassingly parallel, or EP. In [1, p. 31, bullet 3], this point is made for the EP benchmark. Also, on page 29 of [1], Bailey describes the binary algorithm for

exponentiation that allows one to compute $a^n \bmod 2^k$ in $\log_2(n)$ steps. The fact that $a^n \bmod 2^k$ is computable in $\log_2(n)$ steps is crucial to making the random-number generator (3) embarrassingly parallel. Bailey implicitly points this out in [1, p. 31, bullet 2]. On the IBM POWER2 Model 590, our bit-wise identical algorithms corresponding to Equation (3) compute more than 40 million random numbers per second. On the IBM SP2 machine, using the Model 590 for its nodes, and using p such nodes, we can compute more than $40p$ million random numbers per second because of the EP nature of these algorithms. Thus, using 25 nodes our algorithms can compute more than a billion random numbers in a second.

In [1], Bailey gives a generic algorithm that simulates base 2^{23} multiple-precision arithmetic to compute the s_i's in (3). This algorithm requires 18 floating-point operations and four convert-to-integer operations per random number. We implement the same generator using three multiply–adds. Any pseudorandom s_i from (3) can then be placed in the range $0 < x_i < 1$ by the scaling $x_i = 2^{-k}s_i$ or be put in the range $-1 < x_i < 1$ by computing $x_i = 2^{-k+1}s_i - 1$. These computations respectively require one and two additional floating-point operations. In this paper, we redefine (3) to compute each x_i directly, without first computing s_i, and thus compute

$$x_{i+1} = ax_i \bmod 1.$$

This change avoids the actual scalings done above. We also show how these computations can be done by three and 3.125 multiply–adds per random number on RS/6000 machines.

When doing modular arithmetic on integers, it is natural to use the greatest integer function. In floating-point arithmetic this is achieved by using the IEEE round-to-zero (chop) rounding mode. Throughout this paper, except for one place in Section 2, we use the chop rounding mode.

Many NIC algorithms are rated by their megaflop rate, although this popular measure is often misleading. We think that pseudorandom-number generation is such a case. An NIC computation related to pseudorandom generation is the EP NAS benchmark [1, 6]. By using mathematical innovation and the fast random-number generation described here, we were able to demonstrate that the RS/6000 POWER2 Model 590 could currently outperform a Cray YMP for EP [6]. In this paper we compare the ratio of the computing time for the generic algorithm [1] to the time of our algorithm. In both cases we use the same model of RS/6000. The new algorithm is written entirely in Fortran and is compiled using the XL Fortran (XLF) compiler [4].

We also implemented another generator of the type (3) in which $a = 44\,485\,709\,377\,909$ and $k = 48$. This generator is the RANF( ) pseudorandom function used on the CDC Cyber 174 computer, and is also generator 2 in the RANDOM_NUMBER intrinsic function provided with the IBM XLF and XLHPF compilers. It is described in the book *Stochastic Simulation* by Brian Ripley [7, p. 216], who presents statistical test results for several generators; he finds this generator "quite acceptable." Gordon Slishman has implemented this generator[1] using three multiply–adds for single-processor execution of RANDOM_NUMBER. The parallel implementation in XLF and XLHPF is described in Section 4.

Slishman also implemented the generator (2), for which the modulus is not a power of 2, for generator 1 of single-processor RANDOM_NUMBER. Parallel versions of both generators 1 and 2 in RANDOM_NUMBER were implemented by Robert Enenkel for SP (distributed-memory) machines, and by Enenkel and Xinmin Tian[2] for SMP (shared-memory) machines. Generator (2) is discussed in detail in Section 5.

In Section 2 we describe some elementary mathematical ideas that can be used to compute (3) rapidly. These ideas are used to derive an algorithm to compute pseudorandom numbers in the range (0, 1) in three multiply–adds and pseudorandom numbers in the range (−1, 1) in 3.125 multiply–adds. Also, using the IEEE round-to-nearest mode, we show how the latter computation can be done in three multiply–adds.

We now discuss POWER and PowerPC models. Before the advent of vector processors, integer arithmetic was generally faster than floating-point arithmetic. It was then customary to produce pseudorandom numbers using fixed-point arithmetic and then convert these integers to floating point with scaling to get floating-point pseudorandom numbers. When fast floating-point processors became available, it became more economical to produce the random numbers directly in floating point. For example, the first equation of generator (1) could be computed on an RS/6000 by setting $0 < s_0 < 2^k$, s_0 odd, and letting, for $i \geq 0$,

$$u = \mathrm{fl}(as_i + 2^{52+k}),$$

$$v = u \ominus 2^{52+k},$$

$$s_{i+1} = \mathrm{fl}(as_i - v).$$

The above three statements define pseudorandom integers in the range $0 < s_i < 2^k$ that are bit-wise identical to the generator (1). However, one usually wants random floating-point numbers in the range (0, 1) or (−1, 1). The formulas above would then require modification; i.e.,

$$x_i = 2^{-k}s_i \quad \text{or} \quad x_i = 2^{-k+1}s_i - 1.$$

[1] Gordon Slishman, private communication, IBM Toronto Laboratory, January 1994.
[2] Xinmin Tian, private communication, IBM Toronto Laboratory, 1997.

However, these computations require an extra multiply–add in addition to the four needed to compute s_i. One intention of this paper is to demonstrate that this additional multiply–add can be removed. This results in improvement factors of 4/3 and 4/3.125, which come to 33% and 28% improvements per iteration over computation based on the above four statements.

In Section 3 we describe two full-period linear congruential generators,

$$x_{i+1} = (ax_i + c) \bmod 1, \tag{4}$$

that compute pseudorandom numbers in the range (0, 1) in four multiply–adds for $c = 2^{-k}$ and $c = a2^{-k}$. We show a proof that these generators are EP. The proof involves showing that $(1 + a + \cdots + a^{n-1}) \bmod q$ can be computed in $\log_2(n)$ steps. Our four-multiply–add implementation of these generators requires us to avoid conditional operations in the unrolled inner loop. It turns out that ordinary unrolling via vectorization fails. To overcome this failure, we introduce "generalized unrolling," which becomes possible because the generator is EP.

In [2, Section 3.6, pp. 170–173], Knuth provides a summary on "how to generate random numbers." He recommends using a linear congruential generator,

$$X \leftarrow (aX + c) \bmod q, \tag{5}$$

that satisfies seven properties. However, he states that this class of generator applies primarily to machine-level coding and hence is *not* portable. For IEEE arithmetic, it makes sense to choose $q = 2^k$. The fused multiply–add instruction is provided by many computer architectures, including the IBM RS/6000 POWER and PowerPC, IBM S/390* G5, Intel/HP IA-64 Itanium**, Apple Power Mac**, HP Precision Architecture RISC 2.0 (e.g., HP900/800), and SGI MIPS** R-8000.[3] Thus, within this more restrictive domain, we can propose our four-multiply–add generator as being "portable."

In Section 5 we extend the ideas of the previous sections to the generator (2). The modulus of this generator is $2^{31} - 1$, and not a power of 2 as for the previous generators. To achieve high performance, nontrivial changes to the algorithm are required.

In Section 6 we describe the performance of these algorithms relative to the generic algorithm of [1].

2. Three multiplicative congruential generators

Let a and y be positive integers less than 2^k, where $k \le 52$. Clearly a and y are IEEE numbers. Let $x = 2^{-k}y$ so that $0 < x < 1$. Also, x is an IEEE number. Let $p = ax$. The base-2 (bit) representation of p is $p_1p_2 \cdots p_k.p_{k+1}p_{k+2} \cdots p_{2k}$, where each p_i is 0 or 1. Represent $p = I + F$, where $I = p_1 \cdots p_k$ and $F = .p_{k+1} \cdots p_{2k}$; i.e., I and F are the integer and fractional parts of p. Note that $ax \bmod 1 = F$, that I and F are IEEE numbers, and that $2^{52} \le p + 2^{52} \le 2^{53} - 1$. Let

$$u = \mathrm{fl}(ax + 2^{52}). \tag{6}$$

All 2^{52} positive integers in the range 2^{52} to $2^{53} - 1$ are all of the IEEE numbers in that range. Thus, u in (6) is an integer, and since we are in chop mode, u is the largest integer not exceeding $p + 2^{52}$; i.e., $u = p + I$. (See Lemma 1 for a detailed constructive proof.) Let

$$v = u \ominus 2^{52}. \tag{7}$$

The computation in (7) is done exactly, since v is computed using IEEE arithmetic, in which u, 2^{52}, and $v = I$ are all IEEE numbers. Now let

$$w = \mathrm{fl}(ax - v). \tag{8}$$

On RS/6000, w is the fractional part F of ax because the fused multiply–add instruction always delivers the correct answer when its operands and result are IEEE numbers. Note that $ax - v = F$. Thus,

$$w = ax \bmod 1. \tag{9}$$

We have just proved the following.

Theorem 1

The computation in (6), (7), and (8) above produces the result (9). The value computed by (8) is bit-wise identical to the infinitely precise result (9).

When unrolled, Equations (6), (7), and (8) constitute a three-multiply–add implementation of the random-number generator (3). Let $a = 5^{13}$ and choose $1 < s_0 < 2^k$ with s_0 an odd integer and $k = 46$. Set $x_0 = s_0 2^{-k}$. Then $0 < x_0 < 1$. Note that the seven lower-order bits of x_0 are zero. In fact, each x_i, $i \ge 0$, given by

$$x_{i+1} = ax_i \bmod 1 \tag{10}$$

has this property.

We now discuss some aspects of the Algorithm and Architecture approach [8] and how it relates to unrolling. In Section 1.3 of [1], Bailey et al. describe sample codes for the NAS benchmarks. There were two codes distributed for random-number generation, namely RANDLC and VRANLC; the latter is of interest to this paper. Subroutine VRANLC generates n `REAL*8` uniform pseudorandom numbers in the range (0, 1) by using Equation (3). The documentation states that VRANLC is the standard version designed for scalar or RISC systems. A comment in this code states that the DO loop below in (11) which generates the n uniform pseudorandom numbers is *not* vectorizable.

[3] W. Kahan, private communication, Computer Science Division, University of California, Berkeley, August 2000.

```
      T1 =  R23 * A
      A1 =  AINT (T1)
      A2 =  A - T23 * A1
C
C Generate N results. This loop is not
C                     vectorizable.
C
      DO 120 I = 1, N
C
C Break X into two parts such that
C X = 2^23 * X1 + X2, compute
C Z = A1 * X2 + A2 * X1 (mod 2^23), and then
C X = 2^23 * Z + A2 * X2 (mod 2^46).
C
      T1 =  R23 * X
      X1 =  AINT (T1)
      X2 =  X - T23 * X1
      T1 =  A1 * X2 + A2 * X1
      T2 =  AINT (R23 * T1)
      Z  =  T1 - T23 * T2
      T3 =  T23 * Z + A2 * X2
      T4 =  AINT (R46 * T3)
      X  =  T3 - T46 * T4
      Y(I)= R46 * X
120 CONTINUE                                    (11)
```

We believe the authors' statement about the code in (11). Today's compilers are not able to vectorize this loop. On the other hand, Equation (3) is vectorizable. The Algorithm and Architecture approach to handling Equation (3) would be to rewrite the code so that it performs well when run on some actual machine, e.g., an RS/6000 RISC-type machine. In the present case we would also need to unroll Equations (6), (7), and (8). Please note that the code in (11) can be replaced by the following code, in which $TP52 = 2^{52}$:

```
DO i = 0, n - 1
       u = a * x(i) + TP52
       v = u - TP52
       x(i+1) = a * x(i) - v
ENDDO                                           (12)
```

However, the above code will not execute in three cycles because of pipeline delays. This code is also *not* vectorizable by a compiler. In producing the code in (12), we have used the extra accuracy feature of the fused multiply–add instruction that is present on RS/6000 and PowerPC machines. Also, the code in (12) is equivalent to using Equation (9). A compiler cannot recognize this fact. This is where we use the Architecture feature of our approach. We now show how to "unroll" the code in (12). In doing so, we vectorize the code in (12).

Let $a_j = a^j \bmod 2^k$ for $1 \le j \le m$. Note that, from (3),

$$s_{i+j} = a^j s_i \bmod q$$
$$= a_j s_i \bmod q,$$

so that

$$x_{i+j} = \frac{a_j s_i \bmod q}{q}$$
$$= a_j x_i \bmod 1, \; 1 \le j \le m. \qquad (13)$$

Thus, given x_i and (13), we have defined the next m iterations of (10). This unrolling of (10) by m constitutes using a vector of length m to compute these next m iterations of (10). In effect, RS/6000 machines can be viewed as vector machines with a short vector length. Because RS/6000 machines possess functional parallelism (see [9]), RS/6000 machines have very low vector start-up costs. This fact implies that using a short vector length is not a performance drawback. However, to achieve this vectorization it must be programmed. Now let $m = 4$. Thus, the code in (12) becomes

```
DO    i = 0, n - 4, 4
      u = a * x(i) + TP52
      u2 = a2 * x(i) + TP52
      u3 = a3 * x(i) + TP52
      u4 = a4 * x(i) + TP52
      v = u - TP52
      v2 = u2 - TP52
      v3 = u3 - TP52
      v4 = u4 - TP52
      x(i+1) = a * x(i) - v
      x(i+2) = a2 * x(i) - v2
      x(i+3) = a3 * x(i) - v3
      x(i+4) = a4 * x(i) - v4
ENDDO
```

The above code eliminates most of the pipeline delays. Every target of an FMA in that code is separated by three independent FMAs. However, the last FMA of the loop is followed by the first FMA of the loop with i replaced by $i + 4$; thus, there is no separation between the target $x(i + 4)$ and the target u. In order to eliminate all pipeline delays, we need a more complicated "unrolling" of the code in (12). Consider

```
xi = x(4)
xi3 = x(3)
xi2 = x(2)
xi1 = x(1)
DO i = 0, n - 8, 8
        u4 = a4 * xi + TP52
        u3 = a3 * xi + TP52
        u2 = a2 * xi + TP52
        u = a * xi + TP52
        x(i+3) = xi3
        x(i+4) = xi
        v4 = u4 - TP52
        v3 = u3 - TP52
        v2 = u2 - TP52
        v = u - TP52
        xi0 = a4 * xi - v4
        xi3 = a3 * xi - v3
        x(i+1) = xi1
        x(i+2) = xi2
        xi2 = a2 * xi - v2
        xi1 = a * xi - v
        u4 = a4 * xi0 + TP52
        u3 = a3 * xi0 + TP52
        u2 = a2 * xi0 + TP52
        u = a * xi0 + TP52
        x(i+7) = xi3
        x(i+8) = xi0
        v4 = u4 - TP52
        v3 = u3 - TP52
        v2 = u2 - TP52
        v = u - TP52
        xi = a4 * xi0 - v4
        xi3 = a3 * xi0 - v3
        x(i+5) = xi1
        x(i+6) = xi2
        xi2 = a2 * xi0 - v2
        xi1 = a * xi0 - v
ENDDO                                                    (14)
```

The code in (14) eliminates all pipeline delays and hence should execute at the rate of one pseudorandom number every three multiply–adds. To start the pipeline, we need to precompute outside of the loop the first four pseudorandom numbers, $x(i)$, $i = 1, \cdots, 4$. The code in (14) is unrolled by 8.

Now we demonstrate another feature of the Algorithm and Architecture approach. Suppose we decide to use (13) to unroll by $m = 8$. We have seen that a compiler cannot unroll the present loop (12). We were forced to program the unrolling and to introduce the concept of a "vector instruction" into RS/6000 coding. We now claim that this necessity of having to program a "vector instruction" gives rise to a feature of superscalar machines that vector machines do *not* possess. This feature allows us to produce "customized vector instructions." We remark that this is a part of the Algorithm and Architecture approach, and we now illustrate this concept with the example of generating uniform pseudorandom numbers in the range $(-1, 1)$. A standard implementation requires an extra operation to convert the range from $(0, 1)$ to $(-1, 1)$.

Let $c1 = 2^{53}$ and $c2 = 2^{53} - 1$. Let $0 < s < 2^k$ with s odd and $k = 46$. Let $x = 2^{-k+1}s$. Then $0 < x < 2$. Let, for $i \geq 1$,

$$u = \mathrm{fl}(a_j x + c1), \tag{15}$$

$$v = u \ominus c2, \tag{16}$$

and

$$x_{i+j} = \mathrm{fl}(a_j x - v), \tag{17}$$

where $1 \leq j < 8$. For $j = 8$ we let

$$u = \mathrm{fl}(a_7 x + c1), \tag{18}$$

$$v = u \ominus c1, \tag{19}$$

$$x = \mathrm{fl}(a_7 x - v), \tag{20}$$

and

$$x_{i+8} = x \ominus 1. \tag{21}$$

Equations (15) to (21) define a loop, unrolled by 8, in which the next eight random iterates are computed and also the "seed" x is updated in Equation (20).

The first seven iterations cost three multiply–adds, while the last iteration costs four multiply–adds. The functional parallelism of RS/6000 indicates that this loop will execute in 25 multiply–adds, or 25/8 = 3.125 multiply–adds per iteration.

We close Section 2 by demonstrating a use of the IEEE "round-to-nearest" rounding mode. Let $0 < u_i < 1$ be the x_i computed by Equation (10). For each u_i, let $x_i = 2u_i - 1$. Then $-1 < x_i < 1$. Let $c1 = 2^{53} + 2^k$. Let SEED be the seed x_0 for the u_i computation given by Equation (10):

```
C  use round-to-nearest mode

   x(0) = TWO * SEED - ONE

   DO i = 0, n - 1

          uc = a * x(i) + c1

          v = uc - c1

          x(i+1) = a * x(i) - v

   ENDDO

   SEED = HALF * x(n) + HALF
```

We now prove the following.

Theorem 2

The above code produces results that are bit-wise identical to $x_i = 2u_i - 1$, *where* u_i *is the* x_i *given by (10).*

Proof Let $au_i = I + F$, where $0 < F < 1$, so $u_{i+1} = F$. We want to show $x_{i+1} = 2F - 1$. Let $u = ax_i + c1$. Since a is odd, $a = 2a_1 + 1$ for some a_1, and $r = 2(I - a_1) + c1 + 2F - 1$. Note that $-2^k < ax_i < 2^k$, so that $2^{53} < u < 2^{53} + 2^{k+1}$. For IEEE numbers x with exponent 53, namely those x that satisfy $2^{53} \le x \le 2^{54} - 2$, x is an even integer. For round-to-nearest, the computed value of u, namely uc, is the IEEE number closest to u. We claim $uc = c1 + 2(I - a_1)$. Since uc is an even number with exponent 53, it is an IEEE number. Also, $u - uc = 2F - 1$ or $-1 < u - uc < 1$ as $0 < F < 1$. Thus, uc is the closest IEEE number to u. The fused multiply–add instruction computes v and x_{i+1} exactly. Thus, $v = 2(I - a_1)$ and $x_{i+1} = 2F - 1$.

3. Two full-period generators

We now specify a high-level description of the full-period generators. Recall that the full-cycle generator is a linear congruential generator of the form (5) with $q = 2^k$. We follow the recommendation of Knuth (see [2, p. 171]) and choose the value of c to be 1 or a. These two choices of c give rise to the two versions of our full-period generator. These generators have implementations that are nearly the same as the three-multiply–add generator given by Equations (6), (7), and (8).

Let x be the current iterate, in which $0 < x < 1$. Then the following four multiply–add statements in (22) to (25) below describe the full-period generator:

$$u = \mathrm{fl}(ax + 2^{52}), \tag{22}$$

$$v = u \ominus 2^{52}, \tag{23}$$

$$w = \mathrm{fl}(ax - v), \tag{24}$$

$$x = w \oplus \bar{c}. \tag{25}$$

In (25), the last operation, $x = w \oplus \bar{c}$, computes the next random number. In (25), $\bar{c} = 2^{-k}$ when $c = 1$, and $\bar{c} = a2^{-k}$ when $c = a$. We first consider the case $\bar{c} = 2^{-k}$. We were not able to reduce the number of multiply–adds to three as we did in Equations (15) to (21). We would need to define $c2 = 2^{52} + 2^{-46}$, and this value cannot be represented in a double-precision word. In Equation (25), note that $w < 1$. This fact follows from (9). Since $\bar{c}$ is the smallest random number, we are guaranteed that the new $x \le 1$. In fact, x will equal 1 only once in 2^k iterations of the generator. For $k = 46$, this is a very rare event. Suppose $x = 1$. Then (22), (23), and (24) compute $w = 0$ because x is an integer. However, the new $x = \bar{c}$, and we reach the conclusion that the generator is self-correcting! This feature is very important (see [3] for another example), because it is not necessary to "check" x during each iteration of the calculation. Suppose we chose $2^{-46} < \bar{c} < 1$. Then, x in (25) could be greater than 1. In that case we would have to test and possibly correct x during every iteration:

```
IF (x .gt. 1) x = x - 1                    (26)
```

A conditional statement of this sort, when present in a pipelined inner loop, can significantly degrade performance. Thus, the choice of $\bar{c} = 2^{-k}$ is essential to making the performance of the full-cycle generator fast.

To obtain a new random number every four multiply–adds, it is necessary to unroll (22), (23), (24), and (25) by at least a factor of 2. Our numerical experiments on POWER2 showed that unrolling by 8 was necessary. The code for a linear congruential generator without unrolling is

```
DO i = 0, n - 1

       u = a * x(i) + TP52

       v = u - TP52

       w = a * x(i) - v

       x(i+1) = w + c̄

ENDDO                                      (27)
```

Because of pipeline delays, this code will *not* execute at the rate of one random number produced every four multiply–adds. It turns out that unrolling via "program vectorization" described in Section 2 does not work well. To see this, note that

$$x_{j+i} = a_j x_i + \bar{c}_j, \tag{28}$$

where $a_j = a^j \bmod 2^k$ and $\bar{c}_j = [(1 + a + \cdots + a^{j-1}) \bmod 2^k]\bar{c}$. The $\bar{c}_j$'s are now variable! We have previously seen that the choice of $\bar{c} = 2^{-k}$ was essential in order to maintain good performance. Any other value of $\bar{c}$ gave rise to the use of conditional statements in the code, such as the statement in (26). Thus, the type of unrolling that comes from ordinary vectorization fails to provide peak performance. However, another form of unrolling works. In the present context, we call this "generalized unrolling."

Let $N = 2n$ be an arbitrary even integer. Suppose that we can compute a_n and $\bar{c}_n$ of Equation (28) in $\log_2(n)$ steps. Then we can unroll the code by 2 in (27) as follows:

```
DO i = 0, n - 1

        u = a * x(i) + TP52

        u1 = a * x(i+n) + TP52

        v = u - TP52

        v1 = u1 - TP52

        w = a * x(i) - v

        w1 = a * x(i+n) - v1

        x(i+1) = w + c̄

        x(n+i+1) = w1 + c̄

ENDDO                                                    (29)
```

The code in (29) constitutes "generalized unrolling." We have divided the vector $x(0{:}N-1)$ into two vectors $x(0{:}n-1)$ and $y(0{:}n-1) = x(n{:}N-1)$ and worked on them independently. This type of unrolling gives rise to a form of single-instruction, multiple-data (SIMD) parallelism on a single processor. In [9] we also exploit this form of SIMD parallelism on POWER2 machines. The crucial point of the code in (29) for the present application is that the two vectors x and y both use the same $\bar{c}$.

We show how to compute $\bar{c}_n$ in $\log_2(n)$ steps. First note that

$$\bar{c}_{j+i} \bmod 1 = (a^j \bmod 2^k)(\bar{c}_i \bmod 1) + \bar{c}_j \bmod 1 \tag{30}$$

and

$$a^{j+i} \bmod 2^k = (a^j \bmod 2^k)(a^i \bmod 2^k) \tag{31}$$

hold for all i and j. We construct a table $= TBL(2, 0{:}46)$ whose ith entries $TBL(1{:}2, i)$ are a^{2^i} and $\bar{c}_{2^i}$, for $0 \le i \le 46$. Note that

$$TBL(1, i+1) = TBL(1, i) * TBL(1, i) \bmod 2^k, \tag{32}$$

$$TBL(2, i+1) = [TBL(1, i) + 1] * TBL(2, i) \bmod 1, \tag{33}$$

along with $TBL(1, 0) = a$ and $TBL(2, 0) = 2^{-46}$, generates the table in 46 steps. We can now, given x_0 and the precomputed table, use Equations (28), (30), and (31) to compute x_n in $\log_2(n)$ steps as follows:

```
   t = x(0)

   n0 = n

   lb = 0

1  nn = n0/2

   IF (n0 .gt. 2 * nn) THEN

       u = t * TBL (1, lb) + TP52

       v = u - TP52

       w = t * TBL (1, lb) - v

       t = t + TBL (2, lb)

       IF (t .gt. 1) t = t - 1

   ENDIF

   lb = lb + 1

   n0 = nn

   IF (n0 .gt. 0) goto 1

   x(n) = t                                              (34)
```

We now consider the second generator; this generator has $\bar{c} = a2^{-k}$. Consider the following code, in which TPMK $= 2^{-k}$:

```
DO i = 0, n - 1

        u = x(i) + TPMK

        v = a * u + TP52

        w = v - TP52

        x(i+1) = a * u - w

ENDDO                                                    (35)
```

Here we have $0 \le x_i < 1$. When $x_i = 1 - 2^{-k}$, $x_{i+1} = 0$ as $u = 1$. Again, this generator is self-correcting. The unrolled-by-2 version of (35) becomes

```
DO i = 0, n - 1
        u = x(i) + TPMK
        u1 = x(i+n) + TPMK
        v = a * u + TP52
        v1 = a * u1 + TP52
        w = v - TP52
        w1 = v1 - TP52
        x(i+1) = a * u - w
        x(n+i+1) = a * u1 - w1
ENDDO                                                    (36)
```

Equations (30), (31), (32), and (34) remain unchanged. Equation (33) is modified to

$$TBL(2, i+1) = [TBL(1, i) + a] * TBL(2, i) \bmod 1. \quad (37)$$

4. Parallel implementation for HPF

The IBM XL Fortran (XLF) [4] and XL High Performance Fortran (HPF) [5] languages include a RANDOM_NUMBER intrinsic function that implements two multiplicative congruential generators. These generators are also part of PESSL, the IBM Parallel Engineering and Scientific Subroutine Library for AIX [10]. Generator 1 is of type (2), with $a = 7^5$, $k = 31$, and modulus $q = 2^k - 1$. Generator 2 is of type (3), with $a = 44\,485\,709\,377\,909$, $k = 48$, and modulus $q = 2^k$. The generator that is to be used is selected by setting the *generator* argument of RANDOM_NUMBER to 1 or 2, respectively. In this section, we discuss implementation issues arising from the parallel directives in HPF for the modulus 2^{48} generator. The modulus $2^{31} - 1$ generator is discussed in Section 5.

HPF parallel directives

The HPF statement

```
CALL RANDOM_NUMBER (A)
```

fills the scalar, vector, or array A with random numbers. HPF provides directives (ALIGN, DISTRIBUTE, BLOCK, CYCLIC, etc.) that allow the programmer to specify what elements of A are to reside on what processor. To achieve high performance, the parallel algorithm works by computing on each processor only those random numbers resident on that processor, ultimately achieving linear speedups for large quantities of random numbers.

The N random numbers required on a particular processor $j \in \{0, \cdots, P-1\}$ are sub-sequences of $x_0, x_1, x_2, \cdots, x_N$ from (3) of the form

$$x_{ij}, x_{ij+k}, x_{ij+2k}, \cdots, x_{ij+(N-1)k}, \quad (38)$$

where k is called the stride of the sequence. If A is BLOCK-distributed, $k = 1$ and $i_j = jN$, $j = 0, \cdots, P-1$. If A is CYCLIC-distributed, $k = P$ and $i_j = j$, $j = 0, \cdots, P-1$. A contiguous sequence is one with stride $k = 1$, and one with $k > 1$ is called strided. (There is also a BLOCK-CYCLIC distribution and other variations which are not discussed here for brevity, but which are handled by applying techniques similar to those presented here.)

The rest of this section shows how it is possible to compute (38) in $O(N + \log i_j)$ time, resulting in asymptotically linear speedup. The algorithm used depends on whether the required sequence is contiguous or strided.

Contiguous random-number sequences

A contiguous random-number sequence has the form

$$x_i, x_{i+1}, x_{i+2}, \cdots, x_{i+N-1}.$$

Contiguous sequences are required in HPF on each processor when the random numbers are written to a BLOCK-distributed vector or array. Once the starting number x_i is known for a particular processor, the rest of the sequence can be directly computed by the code (14). The fast calculation of the starting numbers on each processor is dealt with next.

Strided random-number sequences

If the CYCLIC distribution directive is used in HPF with more than one processor, the sequence of random numbers resident on each processor consists of strided ($k > 1$) sub-sequences of the form (38). In addition, for a BLOCK distribution, the starting values x_{ij}, $j = 0, \cdots, P-1$ in (38) must be computed from some available previous x_i, for example, x_0. Both of these situations require the efficient computation of multiple steps of the recurrence (13). That is, given x_i and n, we must compute x_{i+n}. The code in (14) does this for several fixed n to achieve the unrolling. However, since n is not known *a priori* in the current context, we cannot simply precompute the value of

$$a_n = a^n \bmod q, \quad (39)$$

and must therefore devise an efficient means to compute it for general n.

We now show how to compute (13) in $O(\log_2 n)$ steps, which is crucial to the asymptotically linear speedup of the parallel algorithm. Binary algorithms for exponentiation are described in [2], and one is used in Bailey's random-

number generator implementation [1]. We use a binary exponentiation algorithm here, but achieve high performance through the use of the FMA instruction.

Let the binary representation of n be

$$n = b_k \cdots b_0,\ b_k \neq 0,\ k = \lfloor \log_2 n \rfloor,\ n \neq 0.$$

Then

$$n = \sum_{j=0}^{k} 2^j b_j$$

and

$$a_n x_i \bmod 1 = [(a^{\sum_{j=0}^{k} 2^j b_j}) \bmod q]\, x_i \bmod 1$$

$$= \left[\left(\prod_{\substack{j=0 \\ b_j \neq 0}}^{k} a^{2^j} \right) \bmod q \right] x_i \bmod 1$$

$$= \left[\prod_{\substack{j=0 \\ b_j \neq 0}}^{k} (a^{2^j} \bmod q) \right] x_i \bmod 1. \tag{40}$$

The values $T_i = a^{2^i} \bmod q$, $i = 0, \cdots, \log_2 q - 1$, are precomputed and stored in a table, allowing the computation of the product (40) in $k \leq \log_2 n$ steps. The product is accumulated in a loop, as in the following pseudocode:

```
t = X(i)

DO i = 0, k

        IF (b_i ≠ 0) THEN

C           Compute t = tT_i mod 1.

            u = t * T(i) + TP52

            v = u - TP52

            t = t * T(i) - v

        ENDIF

ENDDO

X(i+n) = t                                              (41)
```

5. A modulus $2^{31} - 1$ generator

We now consider efficient implementation of the generator of type (2) used in RANDOM_NUMBER. This is an extension of the work in the previous section, but is significantly more complicated since the modulus is no longer a power of 2. For the modulus $2^{31} - 1$ generator, high performance is achieved by scaling the integer recurrence by 2^{-31}, finding a fast implementation of the scaled recurrence, and finally transforming the resulting numbers to the range (0, 1). Both the mathematical issues and the implementation issues arising from the parallel directives in HPF are discussed next.

Generator recurrence

The random-number sequence

$$x_i \in (0, 1),\ i = 0, 1, \cdots \tag{42}$$

produced by the generator is defined by the recurrence

$$s_{i+1} = a s_i \bmod q, \tag{43}$$

$$x_{i+1} = \frac{s_{i+1}}{q}, \tag{44}$$

where s_0 is an integer, $0 < s_0 < q$, and

$a = 7^5 = 16807$,

$q = 2^{31} - 1$.

It has a period of $q - 1$.

This value of q is well suited to an implementation on a machine with 32-bit floating-point arithmetic, such as the IBM S/360 [3]. Although the target architecture of XLF and XLHPF is the IBM RS/6000, which uses IEEE floating-point, this generator has been provided for compatibility with existing applications. The mathematical properties of this generator are discussed in [3].

As for the modulus 2^k generator, the algorithm used depends on whether the required sequence is contiguous or strided.

Contiguous random-number sequences

Before discussing the main contribution of this section, parallel algorithms for strided random-number sequences, it is useful to describe the sequential algorithms (due to G. Slishman[4]) on which they are based. Although the ideas here are based on earlier sections of this paper, the implementation for this generator is more involved, since q is not a power of 2.

RANDOM_NUMBER uses a sequential implementation of (43) and (44) to produce contiguous random-number sequences of the form

$$x_i, x_{i+1}, x_{i+2}, \cdots, x_{i+N-1},$$

when the starting number x_i is known. These sequences are also required by HPF when the random numbers are written to a BLOCK-distributed vector or array.

It is convenient for computational purposes to rewrite (43) and (44) in terms of the value

[4] Gordon Slishman, private communication, IBM Toronto Laboratory, January 1994.

$$y_i = 2^{-31} s_i, \tag{45}$$

so that

$$\begin{aligned} y_{i+1} &= \frac{a y_i 2^{31} \bmod q}{2^{31}} \\ &= \frac{a y_i 2^{31} - \left\lfloor \frac{a y_i 2^{31}}{q} \right\rfloor q}{2^{31}} \\ &= a y_i - \left\lfloor \frac{a y_i 2^{31}}{q} \right\rfloor (1 - 2^{-31}) \\ &= a y_i - \lfloor a y_i s \rfloor (1 - 2^{-31}), \end{aligned} \tag{46}$$

where

$$\begin{aligned} s &= \frac{2^{31}}{q} \\ &= \frac{2^{31}}{2^{31} - 1} \\ &= \frac{1}{1 - 2^{-31}} \\ &= 1 + 2^{-31} + 2^{-62} + 2^{-93} + \cdots . \end{aligned} \tag{47}$$

We now use Lemmas 1, 2, and 6 in the Appendix to transform (46) into an expression (51) that can be efficiently computed with FMAs. Let

$$\bar{s} = 1 + 2^{-31},$$

which is exactly representable as a double-precision IEEE floating-point number. By Lemma 2, we can use $\bar{s}$ in place of s in (48), giving (49). By Lemma 1, we compute $\lfloor a y_i \bar{s} \rfloor$, giving (50). [Since $a y_i \bar{s} = 7^5 2^{-31} s_i (1 + 2^{-31})$ where $s_i < 2^{31} - 1$, $a y_i \bar{s} < 2^{15}$, satisfying the condition of the lemma.] By Lemma 6, we rearrange the terms in the form of three FMAs (51):

$$y_{i+1} = a y_i - \lfloor a y_i s \rfloor (1 - 2^{-31}) \tag{48}$$

$$= a y_i - \lfloor a y_i \bar{s} \rfloor (1 - 2^{-31}) \tag{49}$$

$$= a y_i - (a y_i \bar{s} \oplus 2^{52} \ominus 2^{52})(1 - 2^{-31}) \tag{50}$$

$$= \mathrm{fl}[a y_i - (a y_i \bar{s} \oplus 2^{52})(1 - 2^{-31}) - 2^{52}(1 - 2^{-31})]. \tag{51}$$

The random number x_{i+1} is recovered via (44) and (45) as

$$\begin{aligned} x_{i+1} &= \frac{2^{31} y_{i+1}}{2^{31} - 1} \\ &= s y_{i+1}. \end{aligned}$$

The floating-point values $\tilde{y}_{i+1}$ and $\tilde{x}_{i+1}$, corresponding respectively to y_{i+1} and x_{i+1}, are computed in RANDOM_NUMBER by the sequence of instructions

$$\tilde{y}_0 = y_0$$

and

$$u = \mathrm{fl}(\tilde{y}_i \bar{a} + 2^{52}), \tag{52}$$

$$v = \mathrm{fl}(u k_1 - k_2), \tag{53}$$

$$\tilde{y}_{i+1} = \mathrm{fl}(\tilde{y}_i a - v), \tag{54}$$

$$\tilde{x}_{i+1} = \tilde{y}_{i+1} \otimes \bar{s}, \tag{55}$$

where

$$\bar{a} = a\bar{s},$$

$$k_1 = 1 - 2^{-31},$$

and

$$k_2 = 2^{52} - 2^{21}$$

are exactly representable in IEEE double precision. The program (52)–(55) computes $u = \lfloor \tilde{y}_i \bar{a} \rfloor + 2^{52}$, v is computed exactly, and it follows that $\tilde{y}_{i+1} = y_{i+1}$ and $\tilde{x}_{i+1} = \mathrm{fl}(x_{i+1})$. (A detailed proof is given by Lemma 7 in the Appendix.)

Although an RS/6000 POWER machine is capable of computing one FMA per clock cycle, the code in (52)–(55) will not execute in four cycles because of pipeline delays. The reason for this is that the result of each FMA is not available for 2–3 cycles, although another independent FMA can be started immediately. Pipeline delays can be eliminated by loop unrolling [as explained in Section 2 for the modulus 2^k generator (12)], if a means of efficiently computing y_{i+n}, given y_i, is available. Using (43), (44), and (45), and proceeding as in the derivation of (49), we have

$$\begin{aligned} s_{i+n} &= a^n s_i \bmod q \\ &= a_n s_i \bmod q, \end{aligned}$$

$$y_{i+n} = a_n y_i - \lfloor a_n y_i \bar{s} \rfloor (1 - 2^{-31}), \tag{56}$$

where

$$a_n = a^n \bmod q.$$

This is computed by the following sequence of instructions:

$$\tilde{y}_0 = y_0$$

and

$$u = \tilde{y}_i \otimes \bar{a}_n, \tag{57}$$

$$v = \mathrm{fl}(\tilde{y}_i a_n + u), \tag{58}$$

$$w = v \oplus 2^{52}, \tag{59}$$

$$x = w \ominus 2^{52}, \tag{60}$$

$$y = \mathrm{fl}(\bar{y}_i a_n - x), \tag{61}$$

$$\tilde{y}_{i+n} = \mathrm{fl}(2^{-31}x + y), \tag{62}$$

$$\tilde{x}_{i+n} = \tilde{y}_{i+n} \otimes \bar{s}, \tag{63}$$

where

$$\bar{a}_n = 2^{-31} a_n . \tag{64}$$

The program (57)–(63) computes $x = \lfloor a_n \tilde{y}_i \bar{s} \rfloor$, y is computed exactly, and it follows that $\tilde{y}_{i+1} = y_{i+1}$ and $\tilde{x}_{i+1} = \mathrm{fl}(x_{i+1})$. (A detailed proof is given by Lemma 8 in the Appendix.)

Unlike the modulus 2^k generator, the modulus $2^{31} - 1$ generator requires more instructions to compute a strided random-number sequence than a contiguous one. An unrolled loop based on (57)–(63) takes seven (instead of four) cycles per number on an RS/6000 POWER machine. Therefore, the direct unrolling used for the 2^k generator in (14) does not achieve maximum performance for the $2^{31} - 1$ generator.

Instead, the sequential implementation of the $2^{31} - 1$ generator in RANDOM_NUMBER produces contiguous random-number sequences by using two nested unrolled loops based on (57)–(63) and (52)–(55), analogous to the generalized unrolling in Section 3. For a fixed, preselected batch size $m = 32$, precomputed values of am = a_m, a2m = a_{2m}, a3m = a_{3m}, and abar = $\bar{a}$, ambar = $\bar{a}_m$, a2mbar = $\bar{a}_{2m}$, a3mbar = $\bar{a}_{3m}$ are kept. Given an initial seed $yi = y_i$, the following code (65) then computes random numbers $x(0), \cdots, x(N)$ in batches of size $4m$. (We assume that N is a multiple of $4m$ for simplicity.) It takes $18 + 16m$ cycles per $4m$ random numbers, for an average of 4.14 cycles per number.

This approach is also adopted in the following XLHPF 1.2 parallel algorithm for generating contiguous random-number sequences, which are required when the RANDOM_NUMBER argument vector or array is BLOCK-distributed:

```
  DO i = 0, N - 4 * m, 4 * m
C       (57)-(63) unrolled by 3
        u1 = yi * ambar
        u2 = yi * a2mbar
        u3 = yi * a3mbar
        v1 = yi * am + u1
        v2 = yi * a2m + u2
        v3 = yi * a3m + u3
        w1 = v1 + TP52
        w2 = v2 + TP52
        w3 = v3 + TP52
        x1 = w1 - TP52
        x2 = w2 - TP52
        x3 = w3 - TP52
        y1 = yi * am - x1
        y2 = yi * a2m - x2
        y3 = yi * a3m - x3
        yipm = TPM31 * x1 + y1
        yip2m = TPM31 * x2 + y2
        yip3m = TPM31 * x3 + y3
        DO j = 1, m
C           (52)-(55) unrolled by 4
            u = yi * abar + TP52
            u1 = yipm * abar + TP52
            u2 = yip2m * abar + TP52
            u3 = yip3m * abar + TP52
            v = u * k1 - k2
            v1 = u1 * k1 - k2
            v2 = u2 * k1 - k2
            v3 = u3 * k1 - k2
            yj = yi * a - v
            yjpm = yipm * a - v1
            yjp2m = yip2m * a - v2
            yjp3m = yip3m * a - v3
            x(i+j) = yj * sbar
            x(i+m+j) = yjpm * sbar
            x(i+2 * m+j) = yjp2m * sbar
            x(i+3 * m+j) = yjp3m * sbar
        END DO
        yi = yjp3m
    END DO                                   (65)
```

Strided random-number sequences

If the CYCLIC distribution directive is used in HPF with more than one processor, the sequence of random numbers resident on each processor consists of strided ($k > 1$) sub-sequences of the form (38). In addition, for a BLOCK distribution, the starting values x_{ij}, $j = 0, \cdots, P - 1$, in (38) must be computed from some available previous x_i, for example x_0. Both of these situations require the efficient computation of multiple steps of the recurrence (43), (44). That is, given y_i and n in (56), we must compute y_{i+n}. This is performed by the code in (57)–(63). However, since n is not known *a priori*, we cannot simply precompute the value of

$$a_n = a^n \bmod q,$$

and must therefore devise an efficient means to compute it for general n.

We now consider how to compute a_n in $O(\log_2 n)$ steps, which is crucial to the asymptotically linear speedup of the parallel algorithm. The outline of the approach is similar to (41), but it is complicated by the need to compute the product modulo q instead of 1:

$a_n = 1$

```
DO i = 0, k
        IF (b_i ≠ 0) THEN
            a_n = a_n T_i mod q
        ENDIF
ENDDO                                   (66)
```

A means for efficiently computing the modulus operation in (66) using IEEE double-precision arithmetic and the RS/6000 FMA instruction is a main contribution of this section, and is derived next.

Computing products modulo q

We now consider how to efficiently compute modular products of the form (66). (This is a more detailed description of results that were briefly outlined in [11].) Let

$$d = bc \bmod q,$$

where b and c are integers representable in IEEE double precision (IEEE integers, for short), with $0 < b < q$ and $0 < c < q$, where b and c are powers of a, modulo q. [In particular, the values $b = a_n$ and $c = T(i)$ from (66) satisfy these conditions.] Then

$$d = bc - \left\lfloor \frac{bc}{q} \right\rfloor q.$$

By (47),

$$\frac{1}{q} = 2^{-31} s, \tag{67}$$

so that

$$d = bc - \lfloor bc 2^{-31} s \rfloor (2^{31} - 1).$$

By Lemma 3 in the Appendix, it follows that the infinite series s can be replaced by its first two terms; that is,

$$d = bc - \lfloor bc 2^{-31} \bar{s} \rfloor (2^{31} - 1). \tag{68}$$

[In Lemma 3 we use $m = bc < q^2 < 2^{62}$ and the condition that $q \nmid m$ is satisfied, since $q \nmid b$ and $q \nmid c$ ($b < q$ and $c < q$) and q is prime.]

We now show how (68) can be computed.

Theorem 3

Let b and c be integers representable in IEEE double precision (IEEE integers), with $0 < b < 2^{31} - 1$ and $0 < c < 2^{31} - 1$. Let

$$d = bc \bmod (2^{31} - 1),$$

and compute as follows using IEEE double-precision arithmetic with the round-to-zero rounding mode:

$$u = (2^{-62} b) \otimes c, \tag{69}$$

$$v = \mathrm{fl}[(2^{-31} b)c + u], \tag{70}$$

$$w = v \oplus 2^{52}, \tag{71}$$

$$x = w \ominus 2^{52}, \tag{72}$$

$$y = 2^{31} \otimes x, \tag{73}$$

$$z = \mathrm{fl}(bc - y), \tag{74}$$

$$\tilde{d} = z \oplus x. \tag{75}$$

Then $\tilde{d} = d$.

Proof (See Appendix.)

In RANDOM_NUMBER, strided random-number sequences are computed by an unrolled loop based on (57)–(63), similar to (76) below. Given a stride k, ai = a_{ki}, $i = 1, \cdots, 4$, are first computed by (66) and (69)–(75). Let aibar = $\bar{a}_{ki}$, $i = 1, \cdots, 4$. Starting with y1 = y_{i-3}, y2 = y_{i-2}, y3 = y_{i-1}, yi = y_i, the unrolled loop (76) computes strided random numbers X(i) = x_{ki}, $i = 1, \cdots, N$. It takes seven cycles per random number.

```
DO i = 0, N - 4, 4
        u4 = yi * a4bar
        u3 = yi * a3bar
        u2 = yi * a2bar
        u1 = yi * a1bar
```

```
        X(i+1) = y1 * sbar
        v4 = yi * a4 + u4
        v3 = yi * a3 + u3
        v2 = yi * a2 + u2
        v1 = yi * a1 + u1
        X(i+2) = y2 * sbar
        w4 = v4 + TP52
        w3 = v3 + TP52
        w2 = v2 + TP52
        w1 = v1 + TP52
        X(i+3) = y3 * sbar
        x4 = w4 - TP52
        x2 = w2 - TP52
        x3 = w3 - TP52
        x1 = w1 - TP52
        X(i+4) = yi * sbar
        z4 = yi * a4 - x4
        z3 = yi * a3 - x3
        z2 = yi * a2 - x2
        z1 = yi * a1 - x1
        yi = TPM31 * x4 + z4
        y3 = TPM31 * x3 + z3
        y2 = TPM31 * x2 + z2
        y1 = TPM31 * x1 + z1
END DO                                                    (76)
```

6. POWER2 timing results

We now give timings for the generators described in Sections 2 and 3. We have confined our timing studies to the RS/6000 POWER2 Model 590. POWER2 is capable of producing two FMAs every cycle, with a pipeline delay of two or three cycles [8, 9]. The POWER2 Model 590 has a clock cycle of 15 ns. The code in Equation (14) gives a pipeline delay of two cycles. To be safe we have doubled the unrolling from 8 to 16 in the actual code used for the timing below. (There are enough floating-point registers on the POWER2 to allow one to double the unrolling factor without negative impact. Running at the lesser unrolling, however, did not affect performance in any measurable way.) Since each uniform random number in the range (0, 1) costs three FMAs, the peak possible rate of random-number generation is 44.44 million per second. We measured performance of the (0, 1) generator for batches of double-word random numbers of size $n = 2^i$, where $i = 12$ to 21. The size of the 590 cache is 2^{15} doublewords, and the size of its TLB is 2^{18} doublewords. All timing measurements were done using the XLF RTC (real-time clock) utility, and represent actual elapsed "wall-clock" time, including system time, etc. For i between 14 and 21, the performance was essentially constant. It varied between 42.90 and 43.50 million random numbers per second. For $i = 12$ and 13, the values were 39.33 and 41.79. This dropoff is due to a fixed setup cost for the generator. The setup cost consists of saving and restoring the user rounding mode, setting the rounding mode to chop, initializing the unrolled loop so that stores to memory are optimal, and the completion of the loop modulo the unrolled count. Without the setup cost, we measured a rate of 42.33 million random numbers per second for $i = 12$. For large n this cost becomes negligible. The (0, 1) generator runs at 98% of the peak obtainable rate. The result was independent of a and k. We tested two generators where $a = 5^{13}$ and $k = 46$, and $a = 44\,485\,709\,377\,909$ and $k = 48$.

The generic generator of Bailey et al. [1] ran at the rate of 810 000 pseudorandom numbers for the data in cache and 803 000 for data not in cache. Thus, the new generator is 53 times faster than the generic generator for both data in cache and data not in cache.

We tested two versions of the $(-1, 1)$ generator. For one of these, the number of cycles for 16 random numbers was 25. This is the 3.125-FMA generator, in which rounding is toward zero. The second one produced 16 random numbers in 24 cycles. This is the three-FMA generator, where rounding is to nearest. The results for the round-to-nearest $(-1, 1)$ generator were identical to the results for the (0, 1) generator. Returning to the 3.125-FMA $(-1, 1)$ generator, for $i = 14$ to 21 the performance was essentially constant; it varied between 41.15 and 40.52 million random numbers per second. The peak obtainable rate is 24/25 of 44.44 = 42.67 million random numbers per second. The measured results were at 96.4% of the peak obtainable rate. For $i = 12$ and 13, the rates were 37.53 and 39.91 million random numbers per second. Here again we see the effect of the fixed setup cost. In summary, for large enough batches of numbers the multiplicative random-number generators deliver more than 40 million random numbers per second regardless of whether the batch size fits in cache.

Now we discuss the four-FMA linear congruential generator. Here we chose batches of size 2^i for $i = 12$, 13, 14, and 15. For $i = 12$, 13, and 14, we generated

27.62, 29.35, and 30.14 million random numbers per second. The peak possible rate is 33.33 million per second. Thus, our measured rate is about 90% of the peak obtainable. The smaller value for $i = 12$ was due to the fixed setup cost for this generator. For $i = 15$, the result was 26.40 million random numbers per second. For larger values of i, this rate per second dropped off very sharply because of cache and TLB thrashing. To alleviate this problem, we set $n = 2^i - 544 * 8$. The number 544 is the sum of a page plus a line in doublewords. The factor of 8 was obtained by dividing n by 8 to set up eight different store queues. However, almost any other value of n would have worked equally well. We tried new batches of size n, where $i = 15$ to 21. The rates were 30.32, 29.76, 29.91, 29.76, 29.76, 29.75, and 29.76 million random numbers per second. In summary, the four-FMA generator delivers about 30 million pseudorandom numbers per second both for data in cache and data out of cache.

7. Conclusions

In this paper, we have given several illustrations of a general technique called the Algorithm and Architecture approach [11]. We have used algorithmic innovation and the FMA instruction in the design of several uniformly distributed pseudorandom-number generators for the intervals (0, 1) and (−1, 1).

We have implemented multiplicative congruential pseudorandom-number generators, for the ranges (0, 1) and (−1, 1), of the form

$$s_{i+1} = as_i \bmod 2^k,$$

$$x_{i+1} = 2^{-k}s_{i+1} \quad \text{or} \quad 2^{-k+1}s_{i+1} - 1, \tag{77}$$

which have a period of 2^{k-2}. We have shown that the theoretical cost per random number for the two ranges is respectively 3 and 3.125 multiply–adds on RS/6000 processors. Our codes, on the IBM POWER2 Model 590, run at 98% and 96.4% of the peak obtainable rate, respectively, and produce more than 40 million uniformly distributed pseudorandom numbers per second for both ranges. Additionally, our code sustains the 40 million per second rate for data out of cache. Our code is about 50 times faster than the generic implementation with $k = 46$ given in the NAS parallel benchmarks [1], while producing bit-wise identical results.

We also implemented a linear congruential generator of the form

$$s_{i+1} = (as_i + c) \bmod 2^k,$$

$$x_{i+1} = 2^{-k}s_{i+1}, \tag{78}$$

which has the full period of 2^k. We have shown that the theoretical cost per random number [in the range (0, 1)] for this generator is four multiply–adds on RS/6000 processors. Our code, on the IBM POWER2 Model 590, runs at about 90% of the peak obtainable rate and produces more than 30 million uniformly distributed pseudorandom numbers per second.

Finally, we implemented a multiplicative-congruential generator for the interval (0, 1) of the form

$$s_{i+1} = as_i \bmod (2^k - 1),$$

$$x_{i+1} = \frac{s_{i+1}}{2^k - 1}, \tag{79}$$

for which the modulus is not a power of 2. Generators of type (77) and (79), for the interval (0, 1), are available in the RANDOM_NUMBER intrinsic function of IBM XL Fortran [4] and XL High Performance Fortran [5].

All of the generators reported here are "embarrassingly parallel." Using an IBM SP2 machine with 250 POWER2 wide nodes, it is possible to compute more than ten billion uniform random numbers in a second.

8. Appendix

This section contains detailed proofs for the lemmas used in the paper.

We first prove a result that allows the floor operation to be computed quickly using IEEE arithmetic.

Lemma 1

Let v be representable in IEEE double precision, with $0 \le v < 2^{52}$. Then, with the round-to-zero rounding mode,

$$v \oplus 2^{52} = \lfloor v \rfloor + 2^{52}$$

and

$$(v \oplus 2^{52}) \ominus 2^{52} = \lfloor v \rfloor.$$

Proof The lemma is trivially true for $v = 0$. Therefore, assume that $v > 0$. Since v is IEEE, it can be written in binary as

$$v = b_0 \,.\, b_1 \cdots b_{52} \times 2^e, \; b_0 = 1,$$

where $e < 52$ since $v < 2^{52}$. Then,

$$\lfloor v \rfloor = \begin{cases} b_0 \cdots b_e & e \ge 0, \\ 0 & e < 0. \end{cases}$$

Clearly $\lfloor v \rfloor$ is an IEEE number.

Suppose first that $e \ge 0$. Then $v \oplus 2^{52}$ performs the following addition, the result of which is truncated to 53 bits. The number of zeros inserted for the normalization of v is $k = 52 - e$, and $k \ge 1$ since $e < 52$:

$$\begin{aligned} v &= 0 \,.\, 0 \cdots 0 \;\; b_0 \cdots b_e \;\; b_{e+1} \cdots b_{52} \times 2^{52}, \\ 2^{52} &= 1 \,.\, \qquad\qquad\qquad\quad \times 2^{52}, \\ v \oplus 2^{52} &= 1 \,.\, 0 \cdots 0 \;\; b_0 \cdots b_e \qquad\quad \times 2^{52}. \end{aligned}$$

If $e < 0$, then $k > 52$ and $(v \oplus 2^{52}) = 2^{52}$.

Thus,

$$v \oplus 2^{52} = \lfloor v \rfloor + 2^{52},$$

and this establishes the first part of Lemma 1. Now

$$(v \oplus 2^{52}) \ominus 2^{52} = \lfloor v \rfloor,$$

because the operands and result are IEEE numbers. □

Corollary 1

Let x and y be IEEE double-precision numbers satisfying $0 \le xy < 2^{52}$. *Then* $\mathrm{fl}(xy + 2^{52}) = \lfloor xy \rfloor + 2^{52}$ *and* $\mathrm{fl}(xy + 2^{52}) \ominus 2^{52} = \lfloor xy \rfloor$.

Proof Use Lemma 1, except that $v \oplus 2^{52}$ in the lemma is replaced by $\mathrm{fl}(xy + 2^{52})$, and also

$$\begin{aligned} xy &= 0 \;.\; 0 \cdots 0 \quad b_0 \cdots b_e \quad b_{e+1} \cdots b_{105} \times 2^{52} \\ 2^{52} &= 1 \;. \qquad\qquad\qquad\qquad\quad \times 2^{52} \\ \mathrm{fl}(xy + 2^{52}) &= 1 \;.\; 0 \cdots 0 \quad b_1 \cdots b_e \qquad\quad \times 2^{52}. \end{aligned}$$

Therefore $\mathrm{fl}(xy + 2^{52}) = \lfloor xy \rfloor + 2^{52}$, and the result follows. □

Lemma 2

Let $a = 7^5$, $y_i = 2^{-31} s_i$, *where* s_i *is an integer with* $0 < s_i < q = 2^{31} - 1$, *and* $\bar{s} = 1 + 2^{-31}$, $s = 1 + 2^{-31} + 2^{-62} + \cdots$. *Then* $\lfloor a y_i s \rfloor = \lfloor a y_i \bar{s} \rfloor$.

Proof Let $m = 2^{31} a y_i$. Then Lemma 3 gives the required result, provided we can show that m satisfies the conditions of Lemma 3.

First, we show that m is an integer,

$$m = 2^{31} a y_i = 2^{31} 7^5 2^{-31} s_i = 7^5 s_i, \tag{80}$$

which is an integer because s_i is.

Second, we show that $0 < m < 2^{62}$. From (80), $m > 0$ since s_i is. Also from (80), $m < 7^5 2^{31} < 8^5 2^{31} = 2^{46} < 2^{62}$.

Third, we show that $q \nmid m$. Since q is prime, if $q | m$, then $q | a$ or $q | s_i$. But $q > a$ and $q > s_i$; thus, $q \nmid m$. □

Lemma 3

Let $m \in Z$, $0 < m < 2^{62}$, $(2^{31} - 1) \nmid m$. *Then*

$$\lfloor 2^{-31} m + 2^{-62} m \rfloor = \lfloor 2^{-31} m + 2^{-62} m + 2^{-93} m + 2^{-124} m + \cdots \rfloor.$$

Proof Let the binary representation of m be

$$m = b_1 b_2 \cdots b_{62},$$

where each b_i is either 0 or 1. Then,

$$\begin{aligned} 2^{-31} m &= b_1 \cdots b_{31} \;.\; b_{32} \cdots b_{62}, \\ 2^{-62} m &= \quad . \; b_1 \cdots b_{31} \quad b_{32} \cdots b_{62}, \\ 2^{-93} m &= \quad . \; 0 \quad \cdots 0 \quad b_1 \cdots b_{31} \quad b_{32} \cdots b_{62}. \end{aligned}$$

Let

$$g = 2^{-31} m + 2^{-62} m$$

and

$$\begin{aligned} h_3 &= 2^{-93} m, \\ h_4 &= 2^{-93} m + 2^{-124} m, \\ &\vdots \\ h_i &= \sum_{j=3}^{i} 2^{-31j} m, \qquad i = 3, 4, \cdots, \\ h &= \sum_{j=3}^{\infty} 2^{-31j} m \\ &= \lim_{i \to \infty} h_i \,. \end{aligned} \tag{81}$$

We must show that $\lfloor g + h \rfloor = \lfloor g \rfloor$. Suppose, to get a contradiction, that $\lfloor g + h \rfloor \ne \lfloor g \rfloor$. Then adding h to g must cause the integer part of g to change. Since the 2^{-1} to 2^{-31} bits of h are all zero, this means that either

(A) A carry-out must occur from the 2^{-32} bit of $g + h$, and this carry must be propagated into the integer part;

or

(B) No carry is propagated into the integer part, but all of the (infinite number of) bits in the fractional part of $g + h$ are 1, so that

$$g + h = \lfloor 2^{-31} m \rfloor + .11 \cdots = \lfloor 2^{-31} m \rfloor + 1 \in Z. \tag{82}$$

However, (B) is impossible, since by (47),

$$g + h = \frac{m}{2^{31} - 1},$$

so $(2^{31} - 1) \nmid m$ implies $g + h \notin Z$, contradicting (82). Thus (A) holds. But also,

$$g + h = \sum_{j=1}^{\infty} 2^{-31j} m,$$

and is not an integer. Therefore, for J sufficiently large,

$$\lfloor g + h_J \rfloor = \lfloor g + h \rfloor.$$

Consider the 2^{-31} bit of $g + h_J$. If $b_{62} + b_{31} = 0$ or 10, an incoming carry would change the bit to 1 and no further carry would be propagated. Therefore, by (A), $b_{62} + b_{31} = 1$. Repeating this argument with the higher-order bits shows that

$$b_{i+31} + b_i = 1, \qquad i = 1, \cdots, 31. \tag{83}$$

Now consider the two lowest-order terms in h_J, those corresponding to $j = J - 1$ and $j = J$ in (81):

$$2^{-31(J-1)}m = \ .\ 0 \cdots 0 \quad b_1 \cdots b_{31} \quad b_{32} \cdots b_{62},$$

$$2^{-31J}m \quad = \ .\ 0 \cdots 0 \quad 0 \cdots 0 \quad b_1 \cdots b_{31} \quad b_{32} \cdots b_{62}.$$

There is no carry-out from $b_{32} \cdots b_{62}$ in $2^{-31J}m$, since the corresponding bits in $2^{-31(J-1)}m$ are zero. Therefore, there is no carry-in to the next higher-order 31 bits of the sum, which are therefore all 1 by (83), with no carry-out. Repeating this argument with each higher-order block of 31 bits shows that there is no carry-out from the 2^{-32} bit of $g + h$, contradicting (A). Therefore, $\lfloor g + h \rfloor = \lfloor g \rfloor$. □

Lemma 4
Compute v as in (69) and (70). Then

$$\lfloor v \rfloor = \lfloor bc(2^{-31} + 2^{-62}) \rfloor.$$

Proof Since $b, c \in Z$ with $0 < b < 2^{31} - 1$ and $0 < c < 2^{31} - 1$ implies $bc \in Z$ with $0 < bc < 2^{62}$, we can write bc in binary as

$$bc = .b_1 b_2 \cdots b_{62} \times 2^e,$$

where $1 \le e \le 62$ and $b_1 \neq 0$. Then $2^{-31}bc + 2^{-62}bc$ is the result of the following addition:

$$\begin{aligned}
2^{-31}bc &= b_1 \cdots b_{31} \,.\, b_{32} \cdots b_{62} \times 2^{e-62}, \\
2^{-62}bc &= \ .\ b_1 \cdots b_{31} \quad b_{32} \cdots b_{62} \times 2^{e-62}, \\
2^{-31}bc + 2^{-62}bc &= c_0 \quad c_1 \cdots c_{31} \,.\, c_{32} \cdots c_{62} \quad b_{32} \cdots b_{62} \times 2^{e-62}.
\end{aligned} \tag{84}$$

Since $e - 62 \le 0$,

$$\lfloor 2^{-31}bc + 2^{-62}bc \rfloor = \lfloor c_0 \cdots c_{31} \times 2^{e-62} \rfloor. \tag{85}$$

Now

$$u = 2^{-62}b \otimes c = \ .b_1 \cdots b_{31}\, b_{32} \cdots b_{53} \times 2^{e-62}, \tag{86}$$

since $b_1 \neq 0$ and $b_{54} \cdots b_{62}$ are truncated. The FMA in (70) computes

$$v = \mathrm{fl}(2^{-31}bc + u).$$

When we replace $2^{-62}bc$ in (84) with u from (86), the sum in (84) becomes

$$v = c_0 \cdots c_{31} \,.\, c_{32} \cdots c_{62}\ b_{32} \cdots b_{53} \times 2^{e-62},$$

so that

$$\begin{aligned}
\lfloor v \rfloor &= \lfloor c_0 \cdots c_{31} \times 2^{e-62} \rfloor \\
&= \lfloor 2^{-31}bc + 2^{-62}bc \rfloor
\end{aligned}$$

by (85) as required. □

Lemma 5
Compute y as in (69)–(73). Then $bc - y$ is an IEEE integer.

Proof By (91),

$$bc - y = bc - 2^{31}\lfloor bc(2^{-31} + 2^{-62}) \rfloor \in Z,$$

and so will be IEEE if $|bc - y| < 2^{53}$. For all $x \in R$, $x - 1 < \lfloor x \rfloor \le x$; thus,

$$bc(2^{-31} + 2^{-62}) - 1 < \lfloor bc(2^{-31} + 2^{-62}) \rfloor \le bc(2^{-31} + 2^{-62}),$$

$$\begin{aligned}
bc - 2^{31}bc(2^{-31} + 2^{-62}) &\le bc - 2^{31}\lfloor bc(2^{-31} + 2^{-62}) \rfloor \\
&< bc - 2^{31}[bc(2^{-31} + 2^{-62}) - 1],
\end{aligned}$$

$$-2^{-31}bc \le bc - y < -2^{-31}bc + 2^{31}.$$

However, $0 < bc < 2^{62}$, so

$$-2^{31} < bc - y < -2^{-31} + 2^{31} < 2^{31};$$

hence, $|bc - y| < 2^{31}$. □

Lemma 6
Let $0 \le u < 2^{22}$ be an IEEE number. Then,

$$\begin{aligned}
(u \oplus 2^{52} \ominus 2^{52})(1 - 2^{-31}) = \mathrm{fl}[&(u \oplus 2^{52})(1 - 2^{-31}) \\
&- 2^{52}(1 - 2^{-31})].
\end{aligned} \tag{87}$$

Proof By Lemma 1, $u \oplus 2^{52} = \lfloor u \rfloor + 2^{52}$. Thus, the right side of (87) equals

$$\begin{aligned}
\mathrm{fl}[(\lfloor u \rfloor + 2^{52})(1 - 2^{-31}) - 2^{52}(1 - 2^{-31})] &= \mathrm{fl}[\lfloor u \rfloor (1 - 2^{-31})] \\
&= \mathrm{fl}[\lfloor u \rfloor - 2^{-31}\lfloor u \rfloor].
\end{aligned}$$

Since $0 \le u < 2^{22}$, $\lfloor u \rfloor$ is an integer with at most 22 bits. Therefore, $\lfloor u \rfloor - 2^{-31}\lfloor u \rfloor$ has at most $22 + 31 = 53$ bits, making it exactly representable. The right side of (87) thus becomes $\lfloor u \rfloor (1 - 2^{-31})$, which equals the left side by Lemma 1. □

Lemma 7
(52)–(55) result in $\tilde{y}_{i+1} = y_{i+1}$ and $\tilde{x}_{i+1} = \mathrm{fl}(x_{i+1})$.

Proof By definition, $\tilde{y}_0 = y_0$. Applying induction on i, assume $\tilde{y}_i = y_i$ to show $\tilde{y}_{i+1} = y_{i+1}$. By Corollary 1, $u = \lfloor y_i \bar{a} \rfloor + 2^{52}$. We first show that (53) is exact. $y_i \bar{a} = 2^{-31}s_i a \bar{s} < 2^{-31}(2^{31} - 1)2^{15}(1 + 2^{-31}) < 2^{15}$, so that $u = I + 2^{52}$, where I is an integer less than 2^{15}. Thus, $v = (I + 2^{52})(1 - 2^{-31}) - 2^{52} + 2^{21} = I - 2^{-31}I$. This is an IEEE number, since I is an integer with at most 15 bits, and so v has at most $15 + 31 = 46 < 53$ bits. Therefore, (53) exactly computes

$$v = u(1 - 2^{-31}) - (2^{52} - 2^{21}).$$

Since v is exact, (54) computes $\tilde{y}_{i+1} = y_{i+1}$ exactly by (51), provided y_{i+1} is IEEE. But $y_{i+1} = 2^{-31}s_{i+1}$ by (45),

where s_{i+1} is an integer less than $2^{31} - 1$, so y_{i+1} is an IEEE number.

Finally, to show that (55) computes $\tilde{x}_{i+1} = \mathrm{fl}(x_{i+1})$, we must establish that

$$\mathrm{fl}(y_{i+1}s) = \mathrm{fl}(y_{i+1}\bar{s}). \tag{88}$$

Since s_{i+1} is an integer less than $2^{31} - 1$, we can write $s_{i+1} = b_1 \cdots b_{31}$ in binary. Thus,

$$\begin{aligned} y_{i+1} &= .\ b_1 \cdots b_{31}, \\ 2^{-31}y_{i+1} &= .\ 0\ \cdots 0\quad b_1 \cdots b_{31}, \\ y_{i+1}\bar{s} &= .\ b_1 \cdots b_{31}\ \ b_1 \cdots b_{31}, \\ y_{i+1}s &= .\ b_1 \cdots b_{31}\ \ b_1 \cdots b_{31}\ \ b_1 \cdots b_{31} \cdots. \end{aligned}$$

Let i be the index of the first nonzero bit of y_{i+1}. The 53 bits of the mantissa of $\mathrm{fl}(y_{i+1}s)$ begin at b_i in the first group of 31 bits of $y_{i+1}s$. If $i \le 10$, they end with b_{21+i} in the second group of 31 bits, in which case (88) holds. If $i > 10$, they end with b_{i-10} in the third group of 31 bits. For (88) to hold in this case, we need the bits of the mantissa coming from the third group to be zero, that is, $b_1 = \cdots = b_{i-10} = 0$. But since b_i is the first nonzero bit, $b_1 = \cdots = b_{i-1} = 0$, and the result follows. □

Lemma 8

(57)–(63) result in $\tilde{y}_{i+1} = y_{i+1}$ and $\tilde{x}_{i+1} = \mathrm{fl}(x_{i+1})$.

Proof By definition, $\tilde{y}_0 = y_0$. Applying induction on i, assume $\tilde{y}_i = y_i$ to show $\tilde{y}_{i+1} = y_{i+1}$. We first show that $x = \lfloor v \rfloor = \lfloor y_i a_n \bar{s} \rfloor$. Let $I = s_i a_n$. Then I is an integer satisfying $I < (2^{31} - 1)(2^{31} - 1) < 2^{62}$, so we can write I in binary as $I = b_1 \cdots b_{62}$, and $y_i a_n = 2^{-31}I$. Now $y_i a_n \bar{s} = y_i a_n + y_i \bar{a}_n$, $u = y_i \otimes \bar{a}_n$, $v = \mathrm{fl}(y_i a_n + u)$, and

$$\begin{aligned} y_i a_n &= b_1 \cdots b_{31}\ .\ b_{32} \cdots b_{62}, \\ 2^{-31}y_i a_n &= \qquad\quad .\ b_1\ \cdots b_{31}\quad b_{32} \cdots b_{62}. \end{aligned}$$

For $\lfloor v \rfloor$ to be exact, it is only necessary for the first 31 bits of u to be correct, since the less significant bits align with zeros in $y_i a_n$ and so cannot affect the integer part of the result. But at least the first 53 bits of u are correct; thus, $\lfloor v \rfloor = \lfloor y_i a_n \bar{s} \rfloor$. By Lemma 1, then, $x = \lfloor v \rfloor$.

Next, we show that y is exact. This follows if $y_i a_n - \lfloor y_i a_n \bar{s} \rfloor$ is an IEEE number. $\lfloor y_i a_n \bar{s} \rfloor = \lfloor y_i a_n + 2^{-31} y_i a_n \rfloor = \lfloor y_i a_n \rfloor$ or $\lfloor y_i a_n \rfloor + 1$, since the bits of $2^{-31} y_i a_n$ do not overlap with the integer part of $y_i a_n$. Thus $y_i a_n - \lfloor y_i a_n \bar{s} \rfloor$ is either $b_{32} \cdots b_{62}$ or one less than this value, both of which contain at most 31 bits, and thus are IEEE numbers.

Finally,

$$\begin{aligned} \tilde{y}_{i+1} &= \mathrm{fl}(2^{-31}x + y) \\ &= \mathrm{fl}[2^{-31}\lfloor y_i a_n \bar{s} \rfloor + y_i a_n - \lfloor y_i a_n \bar{s} \rfloor] \\ &= \mathrm{fl}[y_i a_n - \lfloor y_i a_n \bar{s} \rfloor (1 - 2^{-31})] \\ &= \mathrm{fl}[y_i a_n - \lfloor y_i a_n s \rfloor (1 - 2^{-31})] \text{ by Lemma 2} \\ &= \mathrm{fl}(y_{i+n}) \text{ by (56)} \\ &= y_{i+n}, \end{aligned}$$

since $y_{i+n} = 2^{-31} s_{i+n}$ by (45), where s_{i+n} is an integer less than $2^{31} - 1$, so y_{i+n} is an IEEE number.

It follows that $\tilde{x}_{i+n} = \mathrm{fl}(x_{i+n})$, by an argument similar to that used in Lemma 7 to show that $\tilde{x}_{i+1} = \mathrm{fl}(x_{i+1})$. □

Theorem 3

Let b and c be integers representable in IEEE double precision (IEEE integers), with $0 < b < 2^{31} - 1$ and $0 < c < 2^{31} - 1$. Let

$$d = bc \bmod (2^{31} - 1),$$

and compute as in (69)–(75), using IEEE double-precision arithmetic with the round-to-zero rounding mode. Then $\tilde{d} = d$.

Proof From (68),

$$\begin{aligned} d &= bc - \lfloor bc 2^{-31}(1 + 2^{-31}) \rfloor (2^{31} - 1) \\ &= bc - 2^{31} \lfloor bc(2^{-31} + 2^{-62}) \rfloor + \lfloor bc(2^{-31} + 2^{-62}) \rfloor. \end{aligned} \tag{89}$$

We now make use of Lemmas 4 and 5. By Lemma 4,

$$\lfloor v \rfloor = \lfloor bc(2^{-31} + 2^{-62}) \rfloor.$$

Since $bc < 2^{62}$,

$$\begin{aligned} \lfloor v \rfloor &\le \lfloor 2^{62}(2^{-31} + 2^{-62}) \rfloor \\ &= 2^{31} + 1, \end{aligned}$$

so v satisfies the condition of Lemma 1. By Lemma 1,

$$x = \lfloor v \rfloor. \tag{90}$$

Then, since the multiplication in (73) by a power of 2 is exact,

$$\begin{aligned} y &= 2^{31}x \\ &= 2^{31} \lfloor bc(2^{-31} + 2^{-62}) \rfloor. \end{aligned} \tag{91}$$

The FMA in (74) is exact when its result is an IEEE integer. By Lemma 5, $bc - y$ is an IEEE integer; thus, $z = bc - y$. From (89), (90), and (91),

$$\begin{aligned} d &= bc - y + x \\ &= z + x. \end{aligned}$$

Since $z + x = d = bc \bmod q$ is an IEEE integer, it follows that $z + x = z \oplus x = \tilde{d}$. □

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Intel Corporation, Apple Computer, Inc., or MIPS Technologies, Inc.

References

1. D. Bailey, J. Barton, T. Lasinski, and S. Simon, "The NAS Parallel Benchmarks," *Technical Report RNR-91-002*, Revision 2, NASA Ames Research Center, Moffett Field, CA, 1991.
2. D. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, 2nd edition, Volume 2, Addison-Wesley Publishing Co., Reading, MA, 1981.
3. F. G. Gustavson and W. A. Liniger, "A Fast Random Number Generator with Good Statistical Properties," *Computing* **6,** 221–226 (1970).
4. IBM Corporation, *XL Fortran for AIX*, *Language Reference*, *Version 7 Release 1*, 1999, Order No. SC09-2867-00; available through IBM branch offices.
5. IBM Corporation, *XL High Performance Fortran for AIX*, *Language Reference and User's Guide*, *Version 1 Release 3*, 1997; Order No. SC09-2631-00; available through IBM branch offices.
6. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A Very High Performance Algorithm for NAS EP Benchmark," *Proceedings of HPCN'94*, Munich, Germany, April 18–20, 1994.
7. Brian D. Ripley, *Stochastic Simulation*, John Wiley & Sons, Inc., New York, 1987.
8. R. C. Agarwal and F. G. Gustavson, "Algorithm and Architecture Aspects of Producing ESSL BLAS on POWER2, in POWERPC and POWER2: Technical Aspects of the New IBM RISC System/6000," *Technical Report SA23-27-37*, IBM Corporation, 1994.
9. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Exploiting Functional Parallelism of POWER2 to Design High-Performance Numerical Algorithms," *IBM J. Res. & Dev.* **38,** 563–576 (1994).
10. IBM Corporation, *Parallel Engineering and Scientific Subroutine Library for AIX Guide and Reference*, July 2000; Order No. SA22-7273-03; *http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/sp32/pessl/pessl.html*.
11. R. F. Enenkel and F. G. Gustavson, "Fast Calculation of Integer Products Modulo 2**31-1 Using IEEE Floating Point and Fused Multiply–Add," *IBM Tech. Disclosure Bull.* **41** (412), Article 41287 (August 1998).

Received November 1, 2000; accepted for publication September 16, 2001

Ramesh C. Agarwal *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (ragarwal@us.ibm.com).*

Dr. Agarwal received a B.Tech. (Hons.) degree from the Indian Institute of Technology (IIT), Bombay. While there, he received The President of India Gold Medal. He received M.S. and Ph.D. degrees from Rice University and was awarded the Sigma Xi Award for best Ph.D. thesis in electrical engineering. From 1974 to 1977, Dr. Agarwal was at IBM Research in Yorktown Heights, New York. He spent the period 1978–1981 as an Associate Professor at IIT Delhi. In 1982, he returned to IBM. Dr. Agarwal has done research in many areas of engineering, science, and mathematics and has published more than sixty papers in various journals. In 1982, he helped the National Academy of Sciences in studying the acoustic tapes related to the assassination of President Kennedy. He showed that there is no acoustical evidence for the conspiracy theory. In 1994, he analyzed the floating-point divide flaw in the Intel Pentium chip and showed that for spreadsheet calculations using decimal numbers, the probability of divide error increases by several orders of magnitude. His main research focus has been in the area of algorithms and architecture for high-performance computing. His current research activities are in the area of data-mining algorithms and database performance. In 1974, Dr. Agarwal received the Senior Award for best paper from the IEEE ASSP group. He has received several Outstanding Achievement Awards and a Corporate Award from IBM. In 2001, he received a Distinguished Alumnus Award from IIT Bombay. Dr. Agarwal is a Fellow of the IEEE and an IBM Fellow.

Robert F. Enenkel *IBM Toronto Laboratory, Centre for Advanced Studies, M.S. C1/B4R, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (enenkel@ca.ibm.com).*

Dr. Enenkel is a Research Associate at the IBM Centre for Advanced Studies (CAS). Prior to joining CAS, he worked at the IBM Toronto Laboratory on the development of a C compiler and its math library, and developed parallel methods for random-number generation for Fortran and High Performance Fortran compilers. Dr. Enenkel received his B.Sc., M.Sc. and Ph.D. degrees from the University of Toronto, with thesis work in the area of numerical methods for the parallel solution of initial value problems for ordinary differential equations. His current research is in numerical computing as it relates to compilers and operating systems, including floating-point arithmetic, mathematical function libraries, and the performance tuning of algorithms. He is also interested in parallel computing and the application of numerical methods to practical problems in various areas of science. Dr. Enenkel has received an IBM Invention Achievement Award and an IBM Author Recognition Award; he is a member of the Society for Industrial and Applied Mathematics.

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gustav@us.ibm.com).*

Dr. Gustavson manages the Algorithms and Architectures group in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. degree in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-

aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2 for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed and shared memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Outstanding Invention Award, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award. He is a Fellow of the IEEE.

Alok Kothari *Current address not available.*

Mohammad Zubair *Old Dominion University, Computer Science Department, Hampton Boulevard, Norfolk, Virginia 23529 (zubair@cs.odu.edu).* Dr. Zubair has more than thirteen years of research experience in the area of experimental computer science and engineering, both at Old Dominion University and in industry. In his tenure at the University, he has developed several software systems. Two of his research efforts have led to source-code license agreements with major companies. His major industrial assignment was for three years at the IBM Thomas J. Watson Research Center, where he made contributions to IBM ESSL product and in the enabling of parallel benchmarks on IBM SP2. Dr. Zubair's research has been supported by NASA, NSF, ARPA, Los Alamos, AFRL, NRL, JTASC, and the IBM Corporation.