Edgar H. Sibley
Panel Editor

Although superficially time-consuming, on 32-bit computers the minimal-standard random number generator can be implemented with surprising economy.

Two Fast Implementations of the "Minimal Standard" Random Number Generator

David G. Carta

Park and Miller [2] described a number of portable high-level language implementations called the "Minimal Standard" random number generator. These implementations are based on the multiplicative linear congruential form:

$$s_{n+1} = as_n \bmod(2^{31} - 1) \qquad (1)$$

where the currently used multiplier a is 16,807, and the authors project they may switch to $a = 48{,}271$ or $a = 69{,}621$ in the future.

This article presents, in generic form, two implementations at the machine-language level that can be of use when large quantities of random numbers are required and speed of generation is an important issue. This is very frequently the case in real-time simulation.

The discussion is geared to the commonly found computer processors with 32-bit registers that use 1 bit for the sign and 31 bits for the magnitude of integers. We will assume that the product of two integers will occupy two of these registers. Schematically, we present the product as shown in Figure 1.

Each block in Figure 1 represents a 31-bit unsigned integer and $as = 2^{31}p + q$. Note that the quantities a and p occupy less than half their registers.

In the past, a number of generators were implemented with the modulo operation based on 2^m where m is the number of register bits available for the magnitude of an integer. Methods of this type have an enormous advantage in execution time as no actual division—usually a slow operation—needs to be performed. (The original description of the minimal-standard generator [1] used a division.) The integers are multiplied, and the modulo operation is executed by just ignoring the high-order register. By simple manipulation we will show that division is not necessary for calculations modulo $2^{31} - 1$ either.

Using $\mathtt{Frac}$ = Fractional part, we rewrite Equation (1) as

$$s_{n+1} = (2^{31} - 1)\mathtt{Frac}\left[\frac{as_n}{2^{31} - 1}\right] \qquad (2)$$

$$= (2^{31} - 1)\mathtt{Frac}\ as_n[2^{-31} + 2^{-2(31)} + 2^{-3(31)} + \cdots] \qquad (3)$$

$$= (2^{31} - 1)\mathtt{Frac}(2^{31}p + q) \cdot [2^{-31} + 2^{-2(31)} + 2^{-3(31)} + \cdots] \qquad (4)$$

$$s_{n+1} = (2^{31} - 1)\mathtt{Frac}[p + (p + q)2^{-31} + (p + q)2^{-2(31)} + \cdots]. \qquad (5)$$

If $p + q < 2^{31}$, that is, the sum does not overflow its 31-bit register, then the fractional operation above removes just the p, and we arrive at the dramatically simple result

$$s_{n+1} = p + q. \qquad (p + q < 2^{31}). \qquad (6)$$

If $p + q \geq 2^{31}$, then

$$s_{n+1} = (2^{31} - 1)[-1 + (p + q)2^{-31} + (p + q)2^{-2(31)} + \cdots], \qquad (7)$$

so that

$$s_{n+1} = p + q - 2^{31} + 1. \qquad (p + q \geq 2^{31}). \qquad (8)$$

At the machine register level, Equations (6) and (8) have a simple and easily-implemented interpretation: if the sum $p + q$ does not overflow its register, then s_{n+1} is just equal to that sum. If the sum $p + q$ overflows, we simply discard the overflow and increment the register by one to get s_{n+1}.

Though algorithmically different, the procedure just derived is mathematically equivalent to a previous scheme [3] to avoid the division implicit in the modulo operation.

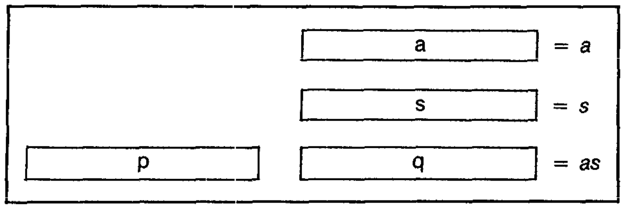

FIGURE 1. A Representation of the Product of Two Integers as Adjacent 31-Bit Registers

AN ALTERNATIVE ALGORITHM

We observe that since p occupies much less than a full register, it will be exceedingly rare that an overflow on $p + q$ will occur. On the average, overflows were observed to occur approximately four times per million iterations.

The infrequent occurrence of an overflow suggested that a useful generator might be implemented by always taking $s_{n+1} = p + q$ and ignoring any overflow. This modified generator is thus equivalent to

$$as_n = 2^{31}p + q, \tag{9}$$

and

$$s_{n+1} = p + q \bmod(2^{31}). \tag{10}$$

The risk of this approach is that the period of the generator will be too short to be useful or that some value of s will become zero, rendering all future values zero. Experimentation with a few starting values produced the useful starting value $s_0 = 40$. With this starting value no overflow occurs until iterate number 1,650,422. Until this first overflow occurs, the sequence is identical to the minimal-standard generator. After $n = 28{,}820{,}115$ iterations (and 101 ignored overflows) s_n reaches the value 6,609 without cycling. At iteration number 56,919,724 the value 6,609 is again reached and further iterations repeat the 28,099,609 long cycle. The dreaded value of zero never occurs.

Effectively, then, with $s_0 = 40$ approximately 57 million iterates are produced before cycling occurs, and the following cycles are about 28 million iterations in length. The starting value $s_0 = 1$ is convenient to remember and almost as good. The first overflow occurs at iteration number 551,246; the value $s_n = 6{,}609$ is reached at iteration step $n = 27{,}427{,}124$; and from there the 28,099,609 iteration cycle begins.

For many purposes the modified procedure can be the generator of choice. It is implemented with a single multiplication and addition at each iteration and requires no logic to determine and correct for overflow. It inherits all of the favorable properties of the minimal generator since between the extremely rare overflows (that are ignored) the sequences are subsequences from the minimal generator. It takes a long time before cycling begins, and once it does, the cycle is also long.

The modified method developed above is of course specific to the multiplier $a = 16{,}807$. The alternative multipliers 48,271 and 69,621 mentioned by Park and Miller in [2] were tested with a few different initial values, and although neither degenerated to zero, neither performed as well as 16,807. With 48,271 the iterations began to cycle at between one and two million steps, and all iterations became trapped in a cycle of about a million steps in length. Using 69,621 as the multiplier with a starting value of 3, the iterations ran for about 15 million steps and then entered into a cycle of about 5 million steps in length.

In the search for longer cycling variations of the modified method, it is suggested that generators with smaller multipliers be examined. By inspection it is obvious that p cannot be larger than a. The smaller p can become, the less likely overflow will take place in the sum $p + q$. With fewer overflows there will be fewer opportunities for cycling. Therefore, testing smaller values of a is indicated.

A SECOND ALTERNATIVE ALGORITHM

A second variation to the basic algorithm also suggests itself. Rather than ignore the occasional overflow of the sum $p + q$ which calls for incrementing the sum by unity, one can simply increment *every* sum, again ignoring overflows. Because this variation has no subsequence belonging to the minimal standard, it was not examined. It might, however, yield a useful fast generator. With a generator of this type, there would be no need to fear a zero appearing in the sequence.

CONCLUDING REMARKS

Using the basic procedure, Equations (6) and (8), the minimal standard can be implemented very economically on computers with 32-bit arithmetic. Depending on the instruction set available and the application using the generator, either of the variations may be viable alternatives.

REFERENCES

1. Lewis, P.A., Goodman, A.S., and Miller, J.M. A pseudo-random number generator for the System/360. *IBM Syst. J. 8*, 2 (1969), 136–146.
2. Park, Stephen K., and Miller, Keith W. Random number generators: Good ones are hard to find. *Commun. ACM 31*, 10 (Oct. 1988), 1192–1201.
3. Payne, W.H., Rabung, J.R., and Bogyo, T.P. Coding the Lehmer pseudorandom number generator. *Commun. ACM 12*, 2 (Feb. 1969), 85–86.

CR Categories and Subject Descriptors: G.3 [**Mathematics of Computing**]: Probability and Statistics—*number generation*; G.4 [**Mathematics of Computing**]: Mathematical Software—*efficiency*
General Terms: Algorithms, Performance, Standardization
Additional Key Words and Phrases: Lehmer generator, machine language, simulation

ABOUT THE AUTHOR:

DAVID G. CARTA is an independent PC consultant with interests in physical, biological, and business applications. He is a licensed control systems engineer. Author's Present Address: CIENTEC, 3775 Fairmeade Road, Pasadena, CA 91107.