Functional Verification of SMP, MPP, and Vector-Register Supercomputers through Controlled Randomness

Joseph T. Wunderlich
Elizabethtown College
Computer Engineering Program

Abstract - Prototype supercomputer functionality can be verified by comparing simulated hardware execution with actual hardware test-program runs where each successive test-program run includes randomly changing machine-states, operating scenarios, and data. Increased verification is achieved through repeated program execution. In both multi-processor and vector-register systems, a "controlled randomness" can be used to verify the functionality of simultaneously executing processors or functional units. This paper discusses the selection and combining of random number generators such that a "degree-of-randomness" between successive or parallel program runs is controlled. This allows computer engineers to simulate the execution of actual software (application or system-level) in which successive or parallel program runs may or may not involve uncorrelated tasks. Additionally, random number generators are selected to maximize execution speed and cycle-length, ensure reproducibility, and when desired, best produce a random source of numbers (i.e., to better approximate an independent, identically-distributed source). Generators can also be chosen for ease of implementation, the ability to run backwards, and the ability to split the generator's cycle into uncorrelated segments. "Backward multipliers" to allow generators to be run in reverse can also be easily found for some types of generators; reversibility is critical for functional verification so that code execution can be traced backwards to find scenarios that led to detected hardware failures. When generators are carefully selected and combined, the verification process can be optimized. By using this methodology, functional verification of SMP, MPP and vector-register supercomputers can be achieved.

TERMS

SMP = Symmetric Multiprocessing
MPP = Massively Parallel Processing
VLSI = Very Large Scale Integration
RNG = Random Number Generator
PASSGEN = RNG's used to randomize machine-states, operating scenarios, and data
PASSGEN() = Function to implement a PASSGEN RNG
SEEDGEN = RNG used to initialize (i.e., seed) PASSGEN's
SEEDGEN() = Function to implement a SEEDGEN RNG
IID = Independent and Identically Distributed
LCG = Linear Congruent Generator
CLCG = Combined Linear Congruent Generator
LFG = Lagged Fibonacci Generator
A = Forward multiplier for LCG's
B = Backward multiplier for LCG's
C = Additive constant for LCG'S
X{I} = Present number generated
X{I-1}= Previous number generated
Q = Special "decomposition" variable for LCG's
R = Special "decomposition" variable for LCG's
M = Modulus
M_CLCG = Modulus for CLCG
J = Lag for LFG'S (the longer one)
K = Lag for LFG'S
X{I-J}= Previous {I-J} seed from LFG seed array
X{I-K}= Previous {I-K} seed from LFG seed array
OPERT= The arithmetic operator used for the LFG (+, or *)
PERIOD = How many numbers generated before sequence repeats (i.e., the "cycle-length")

I. Introduction

Functional verification is part of an overall quality assurance process for computer systems; a process that can include:

1. Functional verification programs run in a simulated prototype-machine environment.
2. Digital and analog VLSI circuit simulation testing.
3. Functional verification programs run on top of a VLSI circuit simulation.
4. Functional verification programs run on prototype hardware.
5. Various instruction-mix and performance benchmark testing.

The idea of using random numbers in test programs has existed for 20 years; however the methodology was typically built on the use of one simple random number generator (RNG). The "controlled randomness" methodology described below allows the combining of six different random number generators for the purpose of creating test programs for functional verification of SMP, MPP and vector-register supercomputers (as well as uni-processor systems):

- For uni-processor implementations, functional verification involves running the same test program(s) repeatedly with different randomization of machine-states, operating scenarios, and data. The "controlled randomness" comes from the correlation between the decisions made, and the data used and changed, within successive program executions.

- For SMP (Symmetric Multiprocessing) systems, each processor can experience the same *temporal* "controlled randomness" as described above, plus an additional *spatial* "controlled randomness" from simultaneously executing test programs on parallel hardware. An example specialized SMP test program is one created to verify cache coherency.

- For MPP (Massively Parallel Processing) systems, the "controlled randomness" is the same as for SMP systems, with the exception of different specialized memory programs to test each processor's associated memory elements.

- For vector-register implementations, the "controlled randomness" exists *spatially* in the parallel functional units executing adjacent vector or matrix elements in a vector operation; and *temporally* from successive test program runs.

The typical test program includes:

1) One or more PASSGEN() functions within each program or parallel thread to randomize machine-states, operating scenarios, and data.

2) Each execution of a program or thread contains an initial random seed generated by a SEEDGEN() function; this is used for the first invocation of a PASSGEN() function in each program pass or parallel thread (i.e., to initialize the PASSGEN). The initial seed for the SEEDGEN can simply be from the real-time clock. Initial seeds are used to identify each program pass or parallel thread.

The testing methodology includes selecting and combining different generators from a set of six that where chosen from a much larger collection [1 to 16]. Each of these six generators is represented by one of three general forms:

1)LINEAR CONGRUENT GENERATORS (LCG)
LCG's are designated as LCG(A,C,M), and have a period equal to M, M/2, M/4, or M/8. LCG's have the form:

```
FORWARD:
X{I}  =(((A)*X{I-1})+C)//M
BACKWARD:
X{I-1}=(((B)*X{I})+C)//M
```

Assuming typical 32-bit computations (i.e., 64-arithematic not available), the intermediate products A*X{} and B*X{} must be kept from creating 32-bit overflow (unless M=2^32 where the //M can just be ignored). This is done by using a "decomposed" form of the above equation (if possible):

```
FORWARD:
 Q=M/A, R=M//A
 IF    A*(X{I-1}//Q) - A*(X{I-1}/R)) > 0
       X{I}= A*(X{I-1}//Q) - A*(X{I-1}/R)
 ELSE
    X{I}=(A*(X{I-1}//Q) - A*(X{I-1}/R))+M

BACKWARD:
  Q=M/B, R=M//B
  IF   (B*(X{I}//Q) - B*(X{I}/R)) > 0
        X{I-1}= B*(X{I}//Q) - B*(X{I}/R)
  ELSE
    X{I-1}=(B*(X{I}//Q) - B*(X{I}/R))+M
```

But this only works if Q > R which is rare (for example, only 23,000 of the 4,000,000,000 32-bit LCG multipliers satisfy this). And finding a LCG with both backward and forward multipliers to satisfy this is even more difficult.

2)COMBINED LINEAR CONGRUENT GENERATORS (CLCG)
CLCG'S are made from two LCG's and have a period of (M1-1)*(M2-1)/2. They have the form:

```
FORWARD:
X{I} = ((LCG(A1,C1,M1) +
         (LCG(A2,C2,M2))//M_CLCG
BACKWARD:
   X{I-1} = ((LCG(B1,C1,M1) +
         (LCG(B2,C2,M2))//M_CLCG
```

3)LAGGED FIBONACCI GENERATORS (LFG)
LFG'S are designated as LFG(J,K,M,OPERTR), and have period of:

```
 ((2^J)-1)*(2^(LOG2(M)-1))
 for the + operator
 ((2^J)-1)*(2^(LOG2(M)-3))
 for the * operator

LFG's have the form:
FORWARD:
  X{I} = (X{I-J} OPERTR X{I-K})//M
NO BACKWARD YET
```

II. Generator Properties

Typically "desirable" generator properties include:

A) Historically proven; has been used for at least several years in industry or academia (i.e., well tested over time).

B) IID; if "good" randomness desired, produces a string of numbers which approximate an independent and identically distributed source (I.I.D.). Independent means the probability of a number being generated is independent of when others generated (i.e., no conditional dependence). Identically distributed means all numbers have an equal probability of being generated (i.e., a uniform distribution). A well-known generator discussed in this paper is "Randu" which is known for poor randomness. This is illustrated in Fig. 1. where every successive three numbers created by the generator (i.e., "three-tuple") is plotted as a point in Cartesian space. A RNG with *good* randomness would show relatively no discernable patterns.

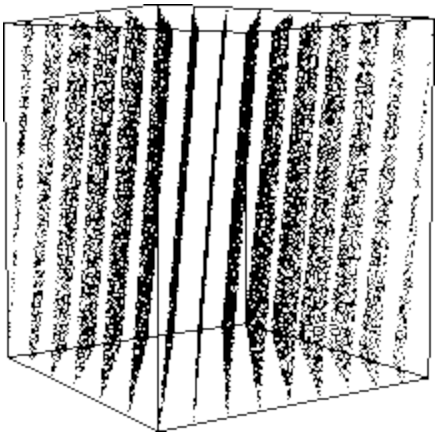

Figure 1. Three-tuple plot of the random number generator "Randu" showing poor randomness.

C) Long period (cycle); (i.e., many numbers produced before generator starts over).

D) Non-overlapping segments; each program pass or parallel thread execution causes a string (a segment) of numbers to be generated by the pass generator (assuming the program or thread contains some PASSGEN() 's). Non-overlapping segments means no significant part of any two segments will be identical; and therefore the generator's period can be broken into non-overlapping segments. This is only possible using the "FIB_A" generator discussed below. However, any generator with a large enough period will most likely produce mostly non-overlapping segments for a typical set of program passes or parallel thread executions. For example, a program with 500 PASSGEN()'s using a segment of 500 numbers; if you run the program for 100,000 passes, you have a total of 50,000,000 numbers used. Even generators with relatively small periods of 500,000,000 would use only 10% of all of the numbers contained within their period. There would be some overlapping segments since the beginning of each segment is chosen randomly at the beginning of each program run -- but possibly not an undesirable amount of overlapping.

E) Execution speed; (both to startup and to run). Programs with many PASSGEN() 's or long PASSGEN() targets (e.g., a large desired string of random data) are referred to as "LONG RUNS" below. Some generators are not well suited for "SHORT RUNS" because of high initialization costs.

F) No repeats of a number within a seed generator's cycle (i.e., period) since a repeating base seed means an identical pass or parallel thread is generated (however, since preceding and following passes or adjacent threads are most likely different, a different scenario may be tested). Repeating numbers are ok for pass generators -- only repeating sequences need to be avoided.

G) Minimal seed memory requirements (i.e., more seeds means more record-keeping and computational overhead).

H) Minimal restrictions on initial seed.

I) Reversibility; The seed generator must go backwards; and the pass generator used by the PASSGEN() 'S is sometimes desired to go backwards. Reversibility is critical for functional verification so that code execution can be traced backwards to find scenarios that led to detected hardware failures.

J) Repeatability is required for debugging. (Note: all of the generators below provide repeatability; both individually and when combined).

III. Evaluation of Seed and Pass Generators

The following seed and pass generators can be specified as part of the functional verification methodology:

- (#1) to (#4) can be used as either a seed or pass generator.
- (#5) and (#6) can only be used as a pass generator since they are not yet reversible.

Generator qualities have been subjectively graded below from (A+) to (F) based on an analysis of algorithm execution times, and an assessment of "spectral data" and other selection criteria from relevant literature [1 to 16]:

1) "RANDU"

```
FORWARD DESIGNATION:  LCG(65539,0,2^32)
BACKWARD DESIGNATION: LCG(477211307,0,2^32)
IID(OF 32-BIT WORDS) .............D
PERIOD............................2^29
OVERLAPPING SEGMENTS..............YES
STARTUP SPEED.....................A+
"SHORT" RUN SPEED.................A+
"LONG" RUN SPEED..................A+
```

REPEATS NUMBER WITHIN PERIOD......NO
NUMBER OF SEEDS..................1
RESTRICTIONS ON INITIAL SEED..NOT 0 OR EVEN
NOTES: Derived from the power residue
method in 1968.

2) "IMPRV"
(AN IMPROVED RANDU-TYPE GENERATOR)

FORWARD DESIGNATION: LCG(71365,0,2^32)
BACKWARD DESIGNATION: LCG(814217229,0,2^32)
IID(OF 32-BIT WORDS) ....B-
PERIOD............................2^29
OVERLAPPING SEGMENTS..............YES
STARTUP SPEED.....................A+
"SHORT" RUN SPEED.................A+
"LONG" RUN SPEED..................A+
REPEATS NUMBER WITHIN PERIOD......NO
NUMBER OF SEEDS..................1
RESTRICTIONS ON INITIAL SEED..NOT 0 OR EVEN

3) "MINSTD"
("MINIMUM-STANDARD" VER. #2)

FORWARD DESIGNATION: LCG(48271,0,(2^31-1))
BACKWARD DESIG.: LCG(1899818559,0,(2^31-1))
IID(OF 32-BIT WORDS) ....B
PERIOD............................2^31
OVERLAPPING SEGMENTS..............YES
STARTUP SPEED.....................A
"SHORT" RUN SPEED.................B
"LONG" RUN SPEED..................B
REPEATS NUMBER WITHIN PERIOD......NO
NUMBER OF SEEDS..................1
RESTRICTIONS ON INITIAL SEED......NOT 0
NOTES:
FORWARD: Using decomposed form to
prevent 32-bit overflow with:
Q=44488,R=3399
BACKWARD: If 64-bit arithmetic is not
available, must use simulated 64-bit
arithmetic to handle 32-bit overflow
since Q is not greater than R for
reverse multiplier.

4) "CLCG"
(COMBINES TWO LCG'S)

GENERATOR #1 FORWARD DESIGNATION:
LCG(40014,0,2147483563)
GENERATOR #1 BACKWARD DESIGNATION:
LCG(2082061899,0,2147483563)
GENERATOR #2 FORWARD DESIGNATION:
LCG(40692,0,2147483399)
GENERATOR #2 BACKWARD DESIGNATION:
LCG(1481316021,0,2147483399)
M_CLCG = 1
IID(OF 32-BIT WORDS) ....B+
PERIOD............................2^63
OVERLAPPING SEGMENTS..............YES
STARTUP SPEED.....................A
"SHORT" RUN SPEED.................B-
"LONG" RUN SPEED..................B-
REPEATS NUMBER WITHIN PERIOD......YES
NUMBER OF SEEDS..................2
RESTRICTIONS ON INITIAL SEED......NOT 0
NOTES:
FORWARD: Using decomposed form to
prevent 32-bit overflow with:
Q1=53668,R1=12211
Q2=52774,R2=3791
BACKWARD: If 64-bit arithmetic is not
available, must use simulated 64-bit
arithmetic to handle 32-bit overflow
since Q is not greater than R for
reverse multiplier.

During initialization, the base seed
created by the SEEDGEN is used as the
initial seed for both constituent
generators.

5) "FIB_M"
(LAGGED FIBONACCI USING MULTIPLICATION)

FORWARD DESIGNATION: LFG(55,24,2^32,*)
BACKWARD DESIGNATION: NOT YET DERIVED
IID(OF 32-BIT WORDS) ....A+
PERIOD............................2^83
OVERLAPPING SEGMENTS..............YES
STARTUP SPEED.....................C+
"SHORT" RUN SPEED.................B-
"LONG" RUN SPEED..................A-
REPEATS NUMBER WITHIN PERIOD......YES
NUMBER OF SEEDS..................55
RESTRICTIONS ON INITIAL SEEDS...SEE NOTES
NOTES: Two seeds of the 55 seeds in the
seed table must be updated each
PASSGEN()invocation, and the seed table
must be initialized for each pass; The
initialization requires filling the seed
table with random values using another
generator, then make all entries odd.

6) "FIB_A"
(LAGGED FIBONACCI USING ADDITION)

FORWARD DESIGNATION: LFG(521,168,2^32,+)
BACKWARD DESIGNATION: NOT YET DERIVED
IID(OF 32-BIT WORDS) ....A
PERIOD............................2^531
OVERLAPPING SEGMENTS..............NO
STARTUP SPEED.....................D
"SHORT" RUN SPEED.................C+

```
"LONG" RUN SPEED.................A-
REPEATS NUMBER WITHIN PERIOD......YES
NUMBER OF SEEDS..................521
RESTRICTIONS ON INITIAL SEEDS...SEE NOTES
NOTES: Two of the 521 seeds in the seed
table seeds must be updated each
PASSGEN()invocation, and the seed table
must be initialized for each pass; The
initialization requires filling the seed
table with random values using another
generator, then to get a unique non-
overlapping segment of the generator's
cycle (i.e., to get the most uncorrelated
program passes or parallel threads), the
initial array must also be put into a
"CANONICAL FORM". This is only possible
for certain J,K pairs and is made by
shifting left (zero into the LSB), clear
the sign bit, then zero the entire last
entry, then the LSB for one or two
special entries is set to one:
          ---------------
          JK-PAIR   ENTRY
          ---------------
           3,2       1
           5,3       2,3
           10,7      8
           17,5      11
           35,2      1
           55,24     12
           71,65     2
           93,91     2,3
           127,97    22
           158,128   64
           521,168   88 (Tested J,K PAIR)
```

IV. Summary of Generators:

	RANDU	IMPRV	MINSTD	CLCG	FIB_M	FIB_A
IID (“randomness”)	D	B-	B	B+	A+	A
PERIOD (cycle)	2^29	2^29	2^31	2^63	2^83	2^531
OVERLAPPING	Y	Y	Y	Y	Y	N
STARTUP SPEED	A+	A+	A	A	C+	D
"SHORT" RUN SPEED	A+	A+	B	B-	B-	C+
"LONG" RUN SPEED	A+	A+	B	B-	A-	A-
REPEATS IN PERIOD	N	N	N	Y	Y	Y
NUMBER OF SEEDS	1	1	1	2	55	521
SEED RESTRICTIONS	not 0, odd	not 0, odd	not 0,	not 0,	MANY	MANY
CAN GO BACKWARDS	Y	Y	Y	Y	N	N

Note: Reverse multipliers where found for the linear congruent generators by simply testing all numbers within each generator’s period (i.e., does one step backwards using a candidate reverse multiplier result in a step equivalent to that of taking one step forward using the forward multiplier.)

V. Controlled Randomness

The “degree-of-randomness” between successive or parallel program runs is controlled through the selection of seed and pass generators. For example,

---

For filling large data areas or for programs with few PASSGEN()’s,
Choose:

SEEDGEN="MINSTD"
PASSGEN="IMPRV"

for very fast, reversible PASSGEN()’s, a single seed, and “ok” randomness; but small period and overlapping segments.

For programs with many PASSGEN()’s (some reversible),
Choose:

SEEDGEN="MINSTD"
PASSGEN="CLCG"

for very random, reversible PASSGEN()’s, and big period; but overlapping segments and two seeds to handle.

For programs with many PASSGEN()’s (none reversible),
Choose:

SEEDGEN="MINSTD"
PASSGEN="FIB_A"

for the ultimate in non-correlated passes or parallel streams (i.e., very good IID and non-overlapping segments); but not reversible PASSGEN()’s and requires 521 seeds.

For any program where ***intentional lack of randomness*** and high correlation between passes or parallel streams is desired,
Choose:

SEEDGEN="RANDU"
PASSGEN="RANDU"

This can often more accurately simulate actual code execution (i.e., lack of randomness and interdependence between successive passes or parallel threads may sometimes be a good thing!).

VI. Conclusions

Prototype SMP, MPP, and vector-register supercomputer functionality can be verified by comparing simulated hardware execution with actual hardware test-program executions where each successive or parallel test-program run includes randomly changing machine-states, operating scenarios, and data. The selection and combining of random number generators, such that a “degree-of-randomness” between program runs or parallel threads is controlled, allows computer engineers to simulate the execution of actual software in which program execution may or may not involve uncorrelated tasks. When generators are carefully selected and combined, verification can be optimized.

References

[1] Niederreiter, H., "**New developments in uniform pseudorandom number and vector generation"**, In Niederreiter, H. and Shiue, P.J.-S., editor(s), Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, volume 106 of Lecture Notes in Statistics. Springer-Verlag, Heidelberg New York, 1995.

[2] Makino, J., "**Lagged-Fibonacci random number generators on parallel computers"**, Parallel Computing, vol. 20, no. 9: pp. 1357-1367, 1994.

[3] Pryor, D. V., et. al., "**Implementation of a Portable and Reproducible Parallel Psuedorandom Number Generator"**, in Proc. of IEEE Int'l Conf. on Supercomputing, pp. 311-319, 1994, Washington, D.C..

[4] Marsaglia, G. and Zaman, A., "**Some portable very-long period random number generators"**, Computers in Physics, vol. 8, no. 1: pp. 117-121. 1994.

[5] Press, W. and Teukolsky, S. A., "**Portable random number generators"**, Computers in Physics, vol. 6, no. 5: pp. 117-121. 1992.

[6] Law, A. M. and Kelton, W. D., "**Simulation modeling and analysis"**, 2nd ed., McGraw-Hill, Boston, MA: 1991.

[7] Anderson, S.L.: "**Random number generators on vector supercomputers and other advanced architectures"**, SIAM Rev., 32: pp. 221-251, 1990.

[8] L'Ecuyer, P., "**Random numbers for simulation"**, Comm. ACM, vol. 33, no.10: pp. 85-97, 1990.

[9] Carter, D. G.: "**Two fast implementations of the "minimal standard" random number generator"**, Comm. ACM, vol. 33, no.1: pp. 87-98, 1990.

[10] Lewis, P.A.W. and Orav, E. J., "**Simulation methods for statisticians, operations analysts and engineers",** Wadsworth & Brooks/Cole, Pacific Grove, CA: 1989.

[11] Maclaren, N. M., "**The generation of multiple independent sequences of pseudorandom numbers"**, J. Appl. Statistics, vol. 38, no.2: pp. 351-359, 1989.

[12] Park, S.K. and Miller, W. M., "**Random number generators: good ones are hard to find"**, Comm. ACM, vol. 31, no.10: pp. 1192-1201, 1988.

[13] Altman, N.S., "**Bit-wise behavior of random number generators"**, SIAM vol. 9, no.5: pp. 941-949, 1988.

[14] L'Ecuyer, P., "**Efficient and portable combined random number generators"**, Comm. ACM, vol. 31, no.6: pp. 85-97, 1988.

[15] Marsaglia, G., "**A current view of random number generators"**, In Billard, L., editor(s), Computer Science and Statistics: The Interface, pp. 3-10. Elsevier Science Publishers B.V., Amsterdam, 1985.

[16] Knuth, D.E., "**The Art of Computer Programming**,", vol. 2: Seminumerical Algorithms. Addison-Wesley, Reading, MA, 2nd edition, 1981.

Dr. JOSEPH T. WUNDERLICH
Dr. Wunderlich is an Assistant Professor of Computer Science and Computer Engineering at Elizabethtown College. Previously, he worked for Purdue University as an Assistant Professor and for IBM as a researcher and hardware development engineer. Dr. Wunderlich received his Ph.D. in Electrical and Computer Engineering from the University of Delaware, his Masters in Engineering Science/Computer Design from The Pennsylvania State University, and his BS in Engineering from the University of Texas at Austin.