Wunderlich, J.T. (2003). **Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness.** In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

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):

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.

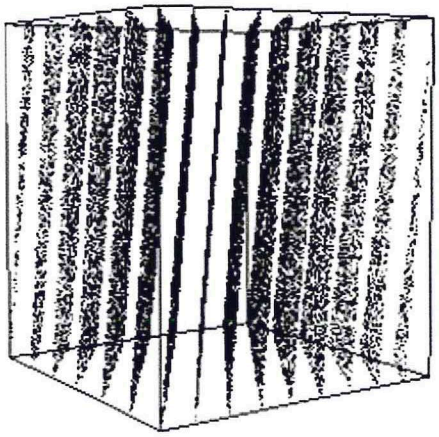


Figure 1. Three-tuple plot of the random number generator "Randu" showing poor randomness.

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).

K) Appropriateness for parallel architectures: Variations of desired randomness within or between processors (or functional units) should be considered [3].

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 19]:

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] Fang, K.-T., Hickernell, F.J., and Niederreiter, H. (editor), **"Monte carlo and quasi-monte carlo methods 2000",** Springer-Verlag, Heidelberg New York, 2002.

[2] Gentle, J. E., **"Random number generation and monte carlo methods (statistics and computing)"**, Springer-Verlag, Heidelberg New York, 1998.

[3] Deng, L.-Y., and Gentle, J. E., **"Lecture 4: parallel random number generation "**, on-line lecture: www.math.ntu.edu.tw/talkdata/021205/1.pdf, 2002.

[4] 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.

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

[6] 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..

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

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

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

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

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

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

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

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

J. T. Wunderlich, Ph.D.
Elizabethtown College
Elizabethtown, PA
Computer Engineering Program

Quality Assurance in Computer Design

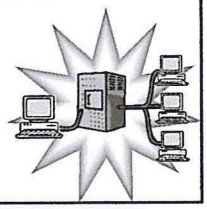
- Functional verification on simulated prototype machine
- Digital & analog VLSI circuit simulation testing
- Functional verification on VLSI circuit simulation
- Functional verification on prototype hardware
- Instruction-mix and performance benchmark testing

Controlled Randomness

- Random numbers in test programs for 20 years
- Method presented: Allow combining six different random number generators

Various Platforms

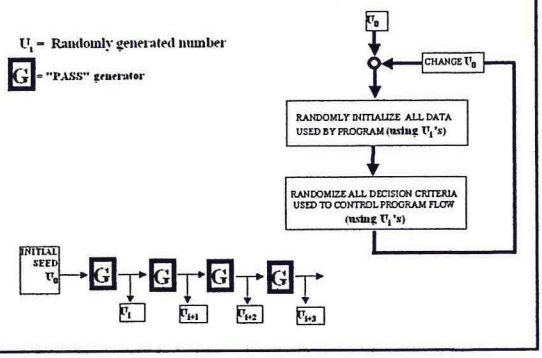
- Uni-processor
- SMP (Symmetric Multiprocessing)
- MPP (Massively Parallel Processing)
- Vector-register



Random Number Generators Terminology

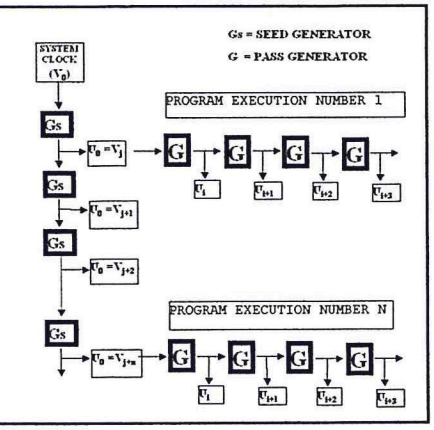
- RNG = Random Number Generator
- PASSGEN = RNG's used to randomize machine-states, operating scenarios, and data
- SEEDGEN = RNG used to initialize (i.e., *seed*) PASSGEN's
- PERIOD = How many numbers generated before sequence repeats (i.e., "cycle-length")

Randomized Programs



PARALLEL

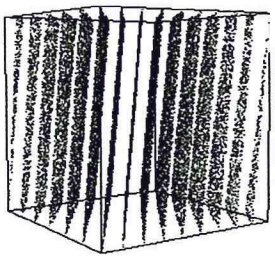
EXECUTION



"Good" Random Number Generator

- Historically proven
- IID (Independent and Idetically Distrbuted)
- Long period (cycle)
- Non-overlapping segments
- Execution speed
- No repeats of a number within seed generator's cycle
- Minimal seed memory
- Minimal restrictions on initial seed
- Reversibility
- Repeatability

"Bad" Random Number Generator



Choosen Generators

- RANDU (Linerar Congruent Generator)
- IMPRV (Linerar Congruent Generator)
- MINSTD (Linerar Congruent Generator)
- CLCG (Combined Linear Congruent Generators)
- FIB_M (Lagged Fibonacci using Multiplication)
- FIB_A (Lagged Fibonacci using Addition)

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

Controlled Randomness

- The "degree-of-randomness" between successive or parallel program runs is controlled through the selection of seed and pass generators

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 intentional lack of randomness and high correlation between passes or parallel streams

- Choose:
  - SEEDGEN="RANDU"
  - PASSGEN="RANDU"
- May more accurately simulate actual code execution (i.e., lack of randomness and interdependence between successive passes or parallel threads is sometimes a good thing!)

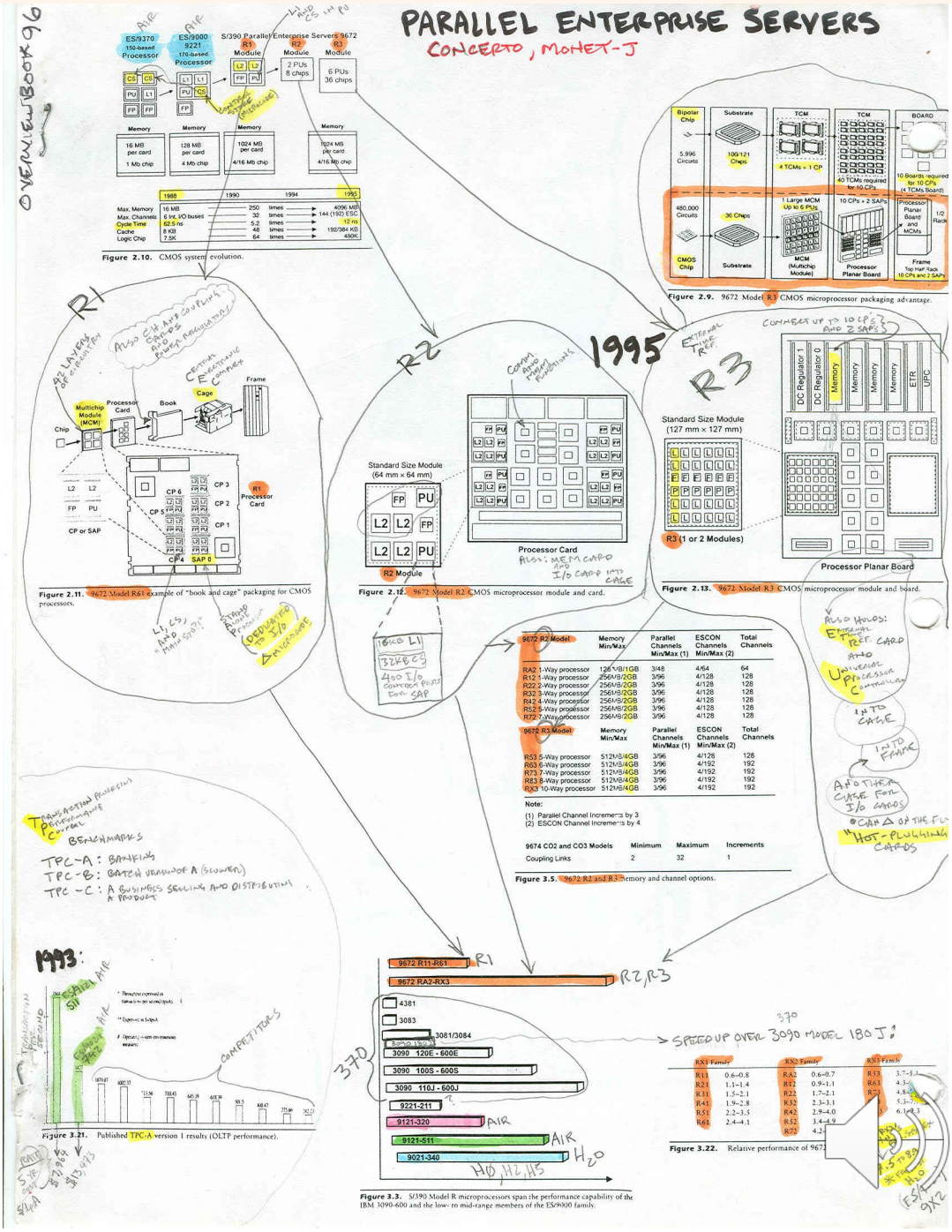
IBM Hardware Development Engineer and Researcher (1996, 97, and 98)
IBM S/390 Hardware Development Lab, Poughkeepsie, NY

- Reviewed specifications for new Symmetric Multi-Processor (SMP) mainframe-supercomputer architectures (jointly developed with IBM Germany) and engineered systems-level software and part of a custom operating system (SAK) to "stress" features and force hardware failures through pseudo-random generation of machine-states and operating scenarios. These SMP machines were designed for up to 20 processors and could be divided into 15 separate logical partitions as well as scaled to 512 processors via a dynamic interconnect facility (IBM Parallel Sysplex). Programs ran in three environments: VLSI circuit simulation, initial hardware test, and manufacturing. 64-bit processing (address and data paths) was introduced during this time requiring simulating 64-bit arithmetic and virtual-address formation to test simulated 64-bit prototype architectures using 32-bit machines; these prototypes were released as the "IBM eServer zSeries" (now called zEnterprise)

- My research included:
  1. Microprocessor branch-prediction verification strategies in a multiprocessor environment.
  2. Random number generator (RNG) theory for hardware verification with seven different correlated random number generators.

- My development projects included creating 20,000 lines of high-level language (PL/X) and S/390 assembly code including operating system application interfaces (API's). My RNG API code was also translated into C for an AIX (IBM's UNIX) environment for IBM AS/400 minicomputers and RS-6000 workstations (the predicessor of POWER7 supercomputers like "Watson") requiring supervision of one engineer in Austin, TX via the IBM intranet. My other development projects included verification programs for cache coherency, virtual addressing, space-switching, linkage control, and 125 new IEEE floating-point instructions (to supplement the existing IBM Hex floating-point instructions). All ~1400 IBM S/390 instructions were tested (including vector-register instructions from previous add-on vector register unit)

- A patent process was initiated for my random number theory and API development.




SUPER COMPUTER DESIGN
The Wonderful Wunderlich
Random Number Generator
HISTORY OF ROME
The Quality Institute
IBM
Exploring IBM's New Age Mainframes
IBM ES/9000
Advanced Computer Architecture

J. Wunderlich was a researcher at IBM before joining Purdue University as an Assistant Professor

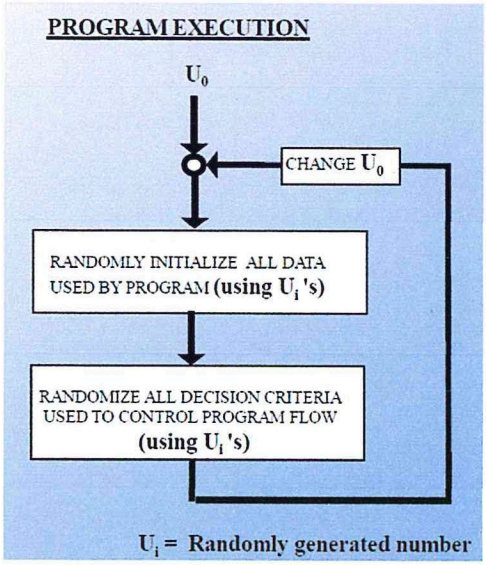
His IBM research was on quality control of S/390 Multi-processor SMP supercomputers

See more here:
http://users.etown.edu/w/wunderjt/home_IBM.html

Quality Assurance in Computer Design

- Functional verification on simulated prototype machine
- Digital & analog VLSI circuit simulation testing
- Functional verification on VLSI circuit simulation
- Functional verification on prototype hardware
- Instruction-mix and performance benchmark testing

Quality Assurance in Computer Design



FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled _ randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

Controlled Randomness

IDEAL GENERATOR

IDEAL GENERATOR

- **(IID) Independent AND Identically Distributed**
- **Identically Distributed: all numbers have equal probability of occurring**
- **Independent: probability of number being generated is independent of when other numbers generated. And therefore, P(A,B, . . . n) = P(A) * P(B) * . . . * P(n)**
- **LONG PERIOD (i.e., numbers generated before repeating)**
- **WELL TESTED**
- **FAST**
- **REPRODUCIBLE**
- **REVERSIBLE**
- **EASILY IMPLEMENTED (machine dependent)**
- **"SPLITTABLE"**

FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled _ randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

RANDOM NUMBER GENERATORS

Programmers have the option of using seven different random number generators for "PASSGEN( ) 'S" (i.e., ?GENBITS, ?GENRNG, ?GENCHAR, ?GENDEC, and ?GENFLOAT); And four different generators for ?GENSEED.

Below is the rationale for which to choose.

TERMINOLOGY:

SEEDGEN= Random number generator used for ?GENSEED (i.e.,the "seed generator" used as the ?GENSEED ALGORITHM)

PASSGEN= Random number generator used for ?GENBITS,?GENRNG, ?GENDEC,?GENCHAR, AND ?GENFLOAT. (i.e.,the "pass generator")

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

FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

X{I}= Present seed

X{I-1}= Previous seed

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

OPERTR= The arithematic operator used for the LFG (+,OR *)

PERIOD= How many numbers generated before sequence repeats (i.e.,the cycle-length)

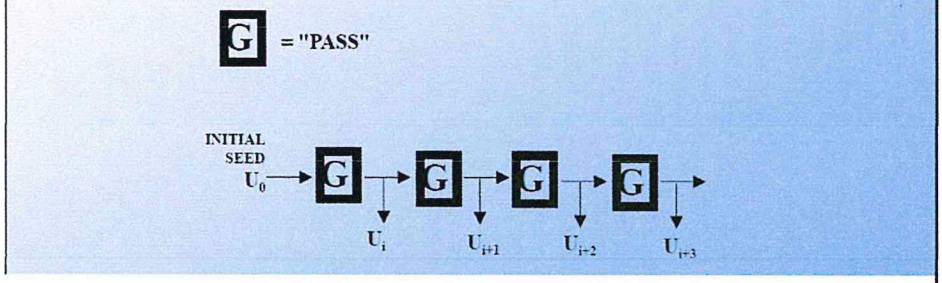
FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

Controlled Randomness

Execution of one program

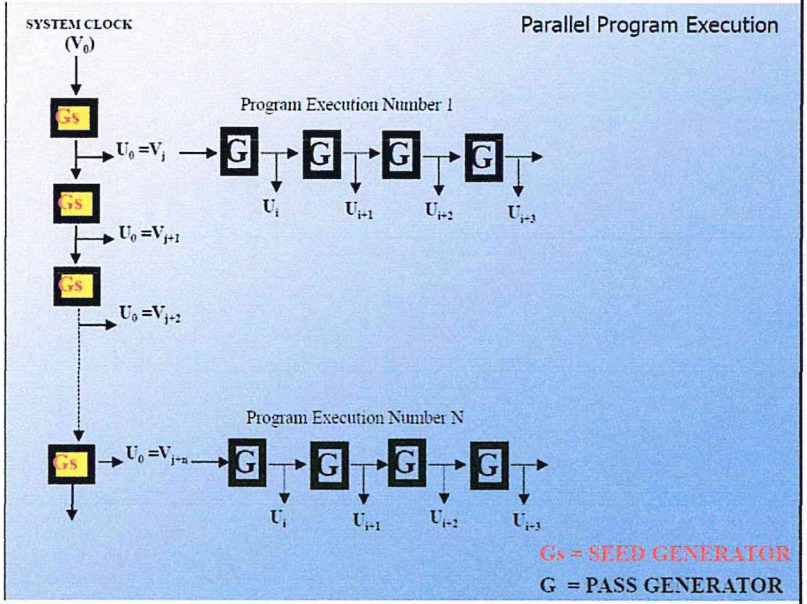


FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica*, M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

Parallel Program Execution



FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica*, M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Controlled Randomness

SEED GENERATOR vs. PASS GENERATOR

SEED GENERATOR vs. PASS GENERATOR

SEED GENERATOR

PERIOD : MAKES NUMBER OF DIFFERENT PASSES. SMALLER FOR MORE PASS CORRELATION.
RANDOMNESS: LESS IMPORTANT THAN FOR PASS GENERATOR. IF DIFFERENT THAN PASS GENERATOR, OVERLAP MINIMIZED.
SPEED: LESS IMPORTANT THAN FOR PASS GENERATOR.
REVERSIBILITY: NEEDED FOR DEBUGGING

PASS GENERATOR

PERIOD: IF EVENLY DIVISIBLE BY NUMBER OF PASS GENERATOR INVOCATIONS IN A PASS, FIRSTPASS WILL REPEAT WHEN PERIOD IS REACHED.
RANDOMNESS: CRITICAL FOR NO CORRELATION BETWEEN PASSES, AND WITHIN PASSES. NO OVERLAP YIELDS BEST RANDOMNESS.
SPEED: MOST IMPORTANT WHEN CREATING LARGE ARRAYS OF RANDOM DATA. INITIALIZATION TIME MORE COSTLY FOR SMALL PROGRAMS.
REVERSIBILITY: USED INFREQUENTLY

FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.
and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Controlled Randomness

SELECTED GENERATORS

SELECTED GENERATORS FOR IBM (by J. Wunderlich, 1997)

SEED GENERATORS

CODE NAME	NUMBER OF SEEDS	PERIOD	RANDOM QUALITY?	SPEED (initial/ running)	CAN GO BACKWARD
OLDGSEED	1	2^26	-	A/B	Y
LCGPRIME (DEFAULT)	1	2^31	B	A/B	Y

PASS GENERATORS

CODE NAME	NUMBER OF SEEDS	PERIOD	RANDOM QUALITY/ OVERLAP?	SPEED (initial/ running)	CAN GO BACKWARD
OLDLCG32 (DEFAULT)	1	2^29	D/Y	A+/A+	Y
NEWLCG32	1	2^29	B-/Y	A/A	Y
COMBOLCG	2	2^63	B+/Y	A-/B-	Y
FIBOMULT	55	2^83	A+/Y	C+/A-	N
FIBOPLUS	521	2^531	A/N	D/A	N

NOTE: ALL GENERATORS WELL TESTED (EXCEPT *OLDGSEED*)
NOTE: FOR *"CONTROLLED RANDOMNESS"*, *OLDGSEED*, *LCGPRIME*, *OLDLCG32*, AND *NEWLCG32* CAN BE SPECIFIED AS BOTH SEED AND PASS GENERATORS

FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.
and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

"CONTROLLED RANDOMNESS"

****

The overall "RANDOM BACKBONE" of a succession of passes can be controlled through the selection of seed and pass generators.

For example,

For filling large data area's or

for programs with few PASSGEN( ) 'S,

Choose: SEEDGEN="MINSTD"

PASSGEN="IMPRV"

for very fast, reversible passes, 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.

FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

For programs with many PASSGEN( ) 'S (none reversible),

Choose: SEEDGEN="MINSTD"

PASSGEN="FIBP"

for the ultimate in non-correlated passes (i.e.,very good word independence and non-overlapping segments); but not reversible PASSGEN( ) 'S and 521 seeds.

For any program where intentional lack of randomness and high correlation between passes is desired,

Choose: SEEDGEN="OGSD"

PASSGEN="OGSD"

OR

Choose: SEEDGEN="RANDU"

PASSGEN="RANDU"

This may closely simulate actual code execution (i.e.,lack of randomness and interdependence between passes may sometimes be a good thing!).

FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design

Controlled Randomness

API CODE EXAMPLES

API's developed by J. Wunderlich, 1997

EXAMPLE USE OF J. Wunderlich API's by System's level programmers:

SEED GENERATOR ("*LCGPRIME*"):

FORWARD: **Gs: $V_i = [(48271 * V_{i-1}) + 0] \bmod (2^{31} - 1)$**
BACKWARD: **Gs: $V_i = [(1899818559 * V_{i-1}) + 0] \bmod (2^{31} - 1)$**

PASS GENERATOR ("*FIBOPLUS*"):

G: $U_i = [U_{i-521} + U_{i-168}] \bmod (2^{32})$

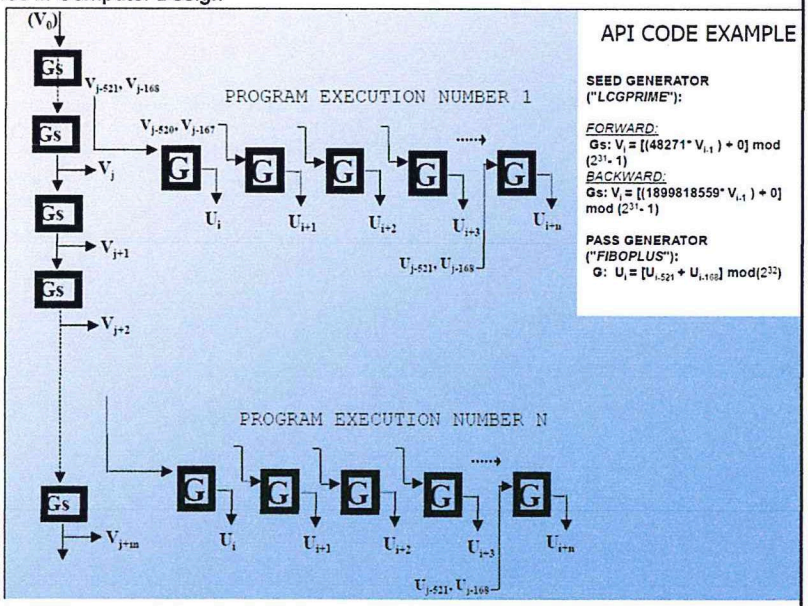
API CODE SYNTAX:

```
?GENSEED [SEEDGEN (XSEEDGEN) ] [PASSGEN (XPASSGEN) ]
PASSGEN ( ) **.....[PASSGEN (XPASSGEN) ]
where *** is BITS,CHAR,DEC,FLOAT,or RNG
```


FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Quality Assurance in Computer Design



API CODE EXAMPLE

SEED GENERATOR ("*LCGPRIME*"):

FORWARD:
Gs: $V_i = [(48271 * V_{i-1}) + 0] \bmod (2^{31} - 1)$
BACKWARD:
Gs: $V_i = [(1899818559 * V_{i-1}) + 0] \bmod (2^{31} - 1)$

PASS GENERATOR ("*FIBOPLUS*"):
G: $U_i = [U_{i-521} + U_{i-168}] \bmod (2^{32})$

FROM: Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

and: Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

Read more here:

Wunderlich, J.T. (2003). Functional verification of SMP, MPP, and vector-register supercomputers through controlled randomness. In *Proceedings of IEEE SoutheastCon, Ocho Rios, Jamaica,* M. Curtis (Ed.): (pp. 117-122). IEEE Press.

Wunderlich, J.T. (1997). Random number generator macros for the system assurance kernel product assurance macro interface. Systems Programmer's User Manual for IBM S/390 Systems Architecture Verification, Poughkeepsie, NY.

RANDOM NUMBER GENERATORS

Programmers have the option of using seven different random number generators for "?GEN*'S" (i.e., ?GENBITS, ?GENRNG, ?GENCHAR, ?GENDEC, and ?GENFLOAT); And four different generators for ?GENSEED. Below is the rationale for which to choose.

TERMINOLOGY:

SEEDGEN=	Random number generator used for ?GENSEED (i.e.,the "seed generator" used as the ?GENSEED ALGORITHM)
PASSGEN=	Random number generator used for ?GENBITS,?GENRNG, ?GENDEC,?GENCHAR, AND ?GENFLOAT. (i.e.,the "pass generator")
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 seed
X{I-1}=	Previous seed
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
OPERTR=	The arithematic operator used for the LFG (+,OR *)
PERIOD=	How many numbers generated before sequence repeats (i.e.,the cycle-length)

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:

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

However, 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). If overflow can't be prevented, 64-bit simulated arithmetic must be used to include the overflow. To prevent overflow, a "decomposed" form of the above equation (if possible) must be used:

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 that satisfy this seems unlikely.

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:

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

LAGGED FIBONACCI GENERATORS (LFG)
****

LFG'S are designated as LFG(J,K,M,OPERTR), and have a period of:

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

LCG's have the form:
X{I} = (X{I-J} OPRERTR X{I-K})//M FORWARD ONLY
NO BACKWARDS YET

GOOD GENERATOR PROPERTIES ARE:

A) Has been used for at least several years in industry or academia (i.e., well tested over time).
B) Produces a string of numbers which approximates 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). For independence, rely on documented testing in the published literature.
For identically distributed, additional testing was done to examine the bit uniformity of each generated 32-bit word for each generator; only the "best" bits are used when using a generator as a pass generator. The entire word is used when using a generator as a seed generator.
C) Long periods (i.e.,want many numbers to be produced before generator starts over). "Cycle" and "period" are synonymous.
D) Can generate non-overlapping segments. Each SAK program pass causes a string (a segment) of numbers to be generated by the pass generator (assuming the program contains some ?GEN*'s). Non-overlapping segments means no significant part of any two segments will be identical. The generator's period can be broken into non-overlapping segments.
This is only possible using the "FIBP" generator. However, any generator with a large enough period will most likely produce mostly non-overlapping segments for a typical set of SAK program passes. For example, a program with 100 ?GEN*'s will use a segment of maybe 500 numbers; and if you run the program 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 their period for this example. There would be some overlapping segments -- but maybe not an undesirable amount. This example assumes relatively small ?GEN* target lengths -- large ?GENBITS targets could lead to many over-lapping segments, but this might be acceptable for some applications.
E) Execution speed (both to startup and to run). SAK programs with many ?GEN*'s or long ?GEN* targets are referred to as "LONG RUNS" below. Some generators are not well suited for "SHORT RUNS" because of high initialization costs.
F) May want no repeats of a number within a seed generator's cycle (i.e.,period) since a repeating base seed means an identical pass is generated (however, since preceding and following passes are most likely different, a different machine state 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 overhead and record keeping).
H) Minimal restrictions on initial seed.
I) Reversibility. The seed generator for SAK must go backwards; and the pass generator used by the ?GEN*'S is sometimes desired to go backwards.
J) Repeatability. This is required for debugging. All of the generators below provide repeatability (both individually and when combined).

The following seed and pass generators can be specified when using ?GENSEED, ?GENBITS, ?GENRNG, ?GENCHAR, ?GENDEC, or ?GENFLOAT.

(#1)to(#4) can be used as either a seed or pass generator.
(#5)to(#7) can only be used for a pass generator since they are not yet reversible.

"MINSTD"(#4) is the default seed generator used by ?GENSEED.
"RANDU" (#2) is the default pass generator used by ?GENBITS,
?GENRNG, ?GENDEC, ?GENCHAR, and ?GENFLOAT (i.e., THE ?GEN*'S).

If (#1,#3,#4,#5,#6,or #7) is specified by ?GENSEED as the default
pass generator, that will be the default for all ?GEN*'S. A
?GEN* can however change the pass generator for one invocation.
Generator qualities have been subjectively graded below
from A+ TO F:

1) "OGSD" (OLD GENSEED)
   ****
   FORWARD DESIGNATION: NONE, IT'S HOME-MADE
   BACKWARD DESIGNATION: NONE, IT'S HOME-MADE
   INDEPENDENCE(OF 32-BIT WORDS).....?
   UNIFORMITY(BITS USED FOR PASSGEN).8:15(1 BYTE)
   PERIOD.....2**26
   OVERLAPPING SEGMENTS.....YES
   STARTUP SPEED.....A
   "SHORT" RUN SPEED.....D
   "LONG" RUN SPEED.....D
   REPEATS NUMBER WITHIN PERIOD.....NO
   NUMBER OF SEEDS.....1
   RESTRICTIONS ON INITIAL SEED.....NOT(0,EVEN,DIVISIBLE BY 5)
   NOTES:
    Derived in the 1970's by someone in SAK to be reversible
    and not create overflow. It was the SAK seed generator for
    25 years. It has the following non-standard form:
     FORWARD:
      IF X{I-1}//2=0 THEN
         X{I}=X{I-1}+'124C41'X
      IF X{I-1}//5=0 THEN
         X{I}=X{I-1}+2
      X{I}=((X{I-1}//1000000000)*31627)//1000000000

     BACKWARD:
      IF X{I-1} IS EVEN THEN
         X{I}=X{I-1}+'124C41'X
      IF X{I-1}//5=0 THEN
         X{I}=X{I-1}+2
      X{I}=((X{I-1}//1000000000)*43222563)//1000000000

2) "RANDU"
   ****
   FORWARD DESIGNATION: LCG(65539,0,2**32)
   BACKWARD DESIGNATION: LCG(477211307,0,2***32)
   INDEPENDENCE(OF 32-BIT WORDS).....D
   UNIFORMITY(BITS USED FOR PASSGEN).8:15(1 BYTE)
   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. It was the
    SAK pass generator for 25 years, and is still the default
    pass generator.

3) "IMPRV" (AN IMPROVED RANDU-TYPE GENERATOR)
   ****
   FORWARD DESIGNATION: LCG(71365,0,2**32)
   BACKWARD DESIGNATION: LCG(814217229,0,2***32)
   INDEPENDENCE(OF 32-BIT WORDS).....B-
   UNIFORMITY(BITS USED FOR PASSGEN).8:15(1 BYTE)
   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

4) "MINSTD" ("MINIMUM-STANDARD" VER. #2)
   ****
   FORWARD DESIGNATION: LCG(48271,0,(2**31-1))

```
   BACKWARD DESIGNATION: LCG(1899818559,(2**31-1))
   INDEPENDENCE(OF 32-BIT WORDS).....B
   UNIFORMITY(BITS USED FOR PASSGEN).8:31(3 BYTES)
   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:
     Using simulated 64-bit arithmetic to handle 32-bit overflow
     since Q->R for reverse multiplier.

    This generator is the default SAK pass generator.
    Minimum Standard versions #2 and #3 are more random than
    version #1. Version #1 is the original Minimum Standard from
    the 1960's. All three versions are in "GENTAB COPY" (with
    backwards multipliers).


5) "CLCG" (COMBINES TWO LCG'S)
   ******************************
   FORWARD DESIGNATION #1:  LCG(40014,0,2147483563)
   BACKWARD DESIGNATION #1: LCG(2082061899,2147483563)
   FORWARD DESIGNATION #2:  LCG(40692,0,2147483399)
   BACKWARD DESIGNATION #2: LCG(1481316021,2147483399)
   INDEPENDENCE(OF 32-BIT WORDS).....B+
   UNIFORMITY(BITS USED FOR PASSGEN).8:31(3 BYTES)
   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=527744,R2=3791
    BACKWARD:
     Using simulated 64-bit arithmetic to handle 32-bit overflow
     since Q->R for reverse multiplier.

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


6) "FIBM" (LAGGED FIBONACCI USING MULTIPLICATION)
   ************************************************
   FORWARD DESIGNATION:  LFG(55,24,2**32,+)
   BACKWARD DESIGNATION: NOT YET DERIVED
   INDEPENDENCE(OF 32-BIT WORDS).....A+
   UNIFORMITY(BITS USED FOR PASSGEN).7:30(3 BYTES)
   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 must be updated in the seed table each ?GEN*
     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.


7) "FIBP" (LAGGED FIBONACCI USING ADDITION)
   ***********************************************
   FORWARD DESIGNATION:  LFG(521,168,2**32,+)
   BACKWARD DESIGNATION: NOT YET DERIVED
   INDEPENDENCE(OF 32-BIT WORDS).....A
   UNIFORMITY(BITS USED FOR PASSGEN).7:30(3 BYTES)
```

```
      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 seeds must be updated in the seed table each ?GEN*
        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), 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 on:
        ---------------
        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 (THIS IS THE J,K PAIR CHOSEN FOR SAK)
```

SUMMARY OF GENERATORS:
****

	OGSD	RANDU	IMPRV	MINSTD	CLCG	FIBM	FIBP
WORD INDEPENDENCE	?	D	B-	B	B+	A+	A
BITS USED (?GEN*)	8:15	8:15	8:15	8:31	8:31	7:30	7:30
PERIOD	2**26	2**29	2**29	2**31	2**63	2**83	2**531
OVERLAPPING	Y	Y	Y	Y	Y	Y	N
STARTUP SPEED	A	A+	A+	A	A	C+	D
"SHORT" RUN SPEED	D	A+	A+	B	B-	B-	C+
"LONG" RUN SPEED	D	A+	A+	B	B-	A-	A-
REPEATS IN PERIOD	N	N	N	N	Y	Y	Y
NUMBER OF SEEDS	1	1	1	1	2	55	521
SEED RESTRICTIONS	MANY	>0,ODD	>0,ODD	>0	>0	MANY	MANY

"CONTROLLED RANDOMNESS"
****

The overall "RANDOM BACKBONE" of a succession of passes can be controlled through the selection of seed and pass generators. For example,

For filling large data area's or
for programs with few ?GEN*'S,
Choose: SEEDGEN="MINSTD"
PASSGEN="IMPRV"
for very fast, reversible passes, a single seed, and ok randomness; but small period and overlapping segments.

For programs with many ?GEN*'S (some reversible),
Choose: SEEDGEN="MINSTD"
PASSGEN="CLCG"
for very random, reversible ?GEN*'S, and big period; but overlapping segments and two seeds to handle.

For programs with many ?GEN*'S (none reversible),
Choose: SEEDGEN="MINSTD"
PASSGEN="FIBP"
for the ultimate in non-correlated passes (i.e.,very good word independence and non-overlapping segments); but not reversible ?GEN*'S and 521 seeds.

For any program where intentional lack of randomness and high

```
      correlation between passes is desired,
         Choose: SEEDGEN="OGSD"
                 PASSGEN="OGSD"
         OR
         Choose: SEEDGEN="RANDU"
                 PASSGEN="RANDU"
         This may closely simulate actual code execution (i.e.,lack
         of randomness and interdependence between passes may
         sometimes be a good thing!).
```