TMS320 DSP
Number 43
DESIGNER'S NOTEBOOK
TEXAS INSTRUMENTS

Random Number Generation on a TMS320C5x

Contributed by Eric Wilbur

Design Problem How is a random number generated on a TMS320C5x?

Solution The philosophy of the term "random" (i.e., how random is random?) has been argued for centuries. I'm sure there were probably several duels held over the years as a result of disagreements on this topic. Some argue that using a computer (a precise, logical, predictable device) to produce random numbers is quite ironic (but useful!). Purists would state that the only truly random event in nature is the time delay between clicks of a Geiger counter placed near a piece of radioactive material.

The goal of this application note is not to solve the ongoing debate over the issue of randomness and somehow vindicate one side or another, but to provide a fast, proven, useful random number generator that can be used in various fixed-point applications.

Theory and Implementation

Many algorithms exist to generate random or pseudo-random numbers. The design objectives of this algorithm were speed, simplicity, "good" results, and the ease of integrating the code into any application. Based on this criteria, a form of uniform deviate called the *linear congruential method* (introduced by D. Lehmer in 1951) was used. The advantages of this method are speed, simplicity to code, and ease of use. However, if care is not taken in choosing the multiplier and increment values, the results can quickly become degenerate. This algorithm produces 65,536 unique numbers and the correlation is very good. Only the LSB exhibits a repeatable pattern every 16 calls.

The linear congruential method has the following form:

Rndnum(n) = (Rndnum(n-l) * MULT) + INC (mod M)

Where:	Rndnum(n)	= current random number
	Rndnum (n- 1)	= previous random number
	Rndnum(l)	= SEED value (arbitrary constant)
	MULT	= multiplier (unique constant)
	INC	= increment (unique constant)
	M	= modulus (word width of 'C5x = 16 bits = 64K)

Much research has been done to identify the optimal choices for the constants MULT and INC. The constants used in this implementation are based on this

research. If changes are made to these numbers, extreme care must be taken to avoid degeneration. Following is a more detailed look at the algorithm and the numbers used:

M: M is the modulus value and is typically defined by the word width of the processor. This algorithm will return a random number between 0 and 65,535 and is NOT internally bounded. If the user requires a min/max limit, this must be coded externally to this routine. The result is not actually divided by 65,536. The accumulator is allowed to overflow, thus implementing the modulus.

SEED: The first random number in the sequence is called the seed value. This is an arbitrary constant between 0 and 64K. Zero can be used, but the first two results of the generator will be 0 and 1. This is OK if the code is allowed 3 calls to "warm up" before the numbers are taken seriously. The number 21,845 was used in this implementation because it is ⅓ of the modulus (65,536).

MULT: Based on random number theory, this number should be chosen such that the last three digits are even-2-1 (such as xx821, x421, etc.). The number 31,821 was used in this implementation. Caution: the generator is extremely sensitive to the choice of this constant!

INC: In general, this constant can be any prime number related to M. Two values were actually tested in this implementation: 1 and 13,849. Research shows that INC should be chosen based on the following formula:

$$\text{INC} = \left(\frac{1}{2} - \left(\frac{1}{6} \times \sqrt{3}\right)\right) \times \text{M} \qquad \text{(Using M=65,536, INC=13,849)}$$

Note: This implementation can be modified to return a 32-bit or 8-bit random number if necessary. For the 32-bit number, simply modify the code to execute a 32×32 multiply instead of 16×16. Remember, your modulus is now 2^32. If an 8-bit result is desired, the low or high byte of the 16-bit result can be used. However, randomness is not guaranteed—duplications will exist.

The Code

```
;************************************************************
;; RANDOM NUMBER GENERATOR FOR THE TMS320C5x DSPs
;;
;; Title:        Randl6.ASM
;; Author.       Eric Wilbur
;; Date:         October 1993
;; Application:  Random Seeks for Hard Disk Drive
;; Target DSP:   TMS320C51
;;
;; Usage:   To Initialize:                Call InitRandl6
;;          To get the next random number: Call _Rand16
;;
;; Assumptions:  SXM,OVM = don't care
;;               SPM = 0 (no shift)
;;
;; Input         None
;; Output        ACCL = 16-bit random number
;************************************************************
```

Figure 1.

```
;;=============================================================
;; MEMORY ALLOCATION
;;=============================================================
;;
Rndnum   .usect  "Variables",1    ;allocate space for random
;;                                ;number result
;;
;;=============================================================
;;INITIALIZE CONSTANTS
;;=============================================================
SEED     .set    21845            ;arbitrary seed value (65536/3)
MULT     .set    31821            ;multiplier value (last 3
                                  ;digits are even-2-1)
INC      .set    13849            ;1 and 13849 have been tested
;;
;;=============================================================
;; CODE START
;;=============================================================
;;
         .text
;;
;;=============================================================
;; INITIALIZE RANDOM NUMBER GENERATOR - Load the SEED value
;;=============================================================
;;
_InitRandl6:      LDP #Rndnum
                  LACC #SEED      ;ACC = SEED value
                  SACL Rndnum     ;Rndnum = SEED
                  RET             ;return to caller
;;
;;=============================================================
;; GENERATE NEXT RANDOM NUMBER
;;=============================================================
;;
_Randl6:          CLRC  OVM       ;clear overflow - implements
                                  ;MOD 64K
                  LDP   #Rndnum   ;set data page pointer
                  LT    Rndnum    ;TReg = Rndnum
                  MPY   #MULT     ;PReg = Rndnum * MULT
                  PAC             ;ACC = Preg
                  ADD   #INC      ;ACC = Rndnum * MULT + INC
                  SACL  Rndnum    ;store new random number
                  RET             ;return to caller
```

Figure 1. (continued)