An tiny, fast, 8-bit pseudo-random number generator in 6502 assembly

by White Flame
(Thanks to bogax for pointing out the $80→$00 link)

This is my re-discovery of the well-known Linear feedback shift register type of PRNG, having seen a bit of its implementation elsewhere.

This simple routine is based on this mutation of a number:

```
        lda seed
        asl
        bcc noEor
         eor #$1d
noEor:  sta seed
```

Shift left, and if the high bit was set, EOR it with a value to twiddle the lower bits as the high ones are lost. As-is, this will cycle through the values $01-$ff exactly once, in a scrambled order, before repeating the same sequence again.

The problem is that setting 'seed' to zero would simply cause it to stay at zero. The high bit is always clear, so the EOR is never applied. If we change the algorithm to perform the EOR when the high bit is clear instead, the pattern is lost and not all numbers are reached, no matter which EOR value is used (yes, I tested all 256 of them). Related to this issue is the fact that the routine never outputs $00, and I need a full range of $00-$ff from my PRNG.

So let's set up a special case for $00, where we will force the EOR to happen:

```
        lda seed
        beq doEor ;added this
         asl
         bcc noEor
doEor:    eor #$1d
noEor:  sta seed
```

This solved the problem of 'seed' being initially set to zero. However, the algorithm will still never output a zero itself. We have 256 handled inputs ($00-$ff) but it is only capable of 255 outputs ($01-$ff).

Looking at the code, we can see that seeds of both $00 and $80 will result in an output of $1d. We need these 2 values to produce unique outputs, so that all of the 256 inputs will yield a unique output, keeping the full period of 256 bytes intact. $00 is already a special input case, but nothing outputs $00. If we make $80 the next special case, and have it output $00, we link our chain into a full 256-cycle loop:

```
        lda seed
        beq doEor
         asl
```

```
         beq noEor ;if the input was $80, skip the EOR
         bcc noEor
doEor:    eor #$1d
noEor:  sta seed
```

And that's it. In other words, these are our 4 cases:

```
 $00 : (shift and) perform EOR
 $80 : just shift

 %1xxxxxxx : shift and perform EOR
 %0xxxxxxx : just shift
```

This yields a chain of all 256 8-bit values in pseudo-random, repeating order. The 256 different possible seeds you can give it will simply start the chain at a different point.

To get a different chain (essentially a different PRNG altogether), the value of the EOR would have to be changed. I ran a loop to test what values would create a chain of all 256 numbers and found 16 of them:

```
$1d (29)
$2b (43)
$2d (45)
$4d (77)
$5f (95)
$63 (99)
$65 (101)
$69 (105)
$71 (113)
$87 (135)
$8d (141)
$a9 (169)
$c3 (195)
$cf (207)
$e7 (231)
$f5 (245)
```

This gives 16 possible PRNGs, each with 256 possible starting seed points. Thus, by specifying 2 bytes as the seed (one for the value of 'seed' and one for the which PRNG's EOR value to use), you can have 4096 different chains of 256 numbers. To be a bit more clever, a 12-bit seed could include the 8-bit 'seed' value and a 4-bit index into the table of 16 possible EOR values. Of course, this is all optional and you could just as well hardcode the EOR instruction to one of the above values to get a consistent PRNG of period 256 that is seeded by a single 8-bit value, as in the above examples.

This algorithm can be extended to 16 or more bits with no problem, especially on architectures of that bit width. The 16-bit version has 2048 EOR values that work, and who knows how many the higher ones have. :)

Check out Small, Fast 16-bit PRNG for a sample implementation of the 16-bit version, and all 2048 available EOR values.