Round 5 • Linear-Feedback Shift Register (LFSR)

OCA / Atlantis

// Introduction

Last time we left off with the challenge of filling the screen randomly. We tested a first idea using a precomputed lookup table (LUT) to hold random numbers, but it consumes 2000 bytes of memory. This time we will use a Linear-Feedback Shift Register (LFSR) to get pseudo-random numbers, avoiding the need for a lookup table completely.

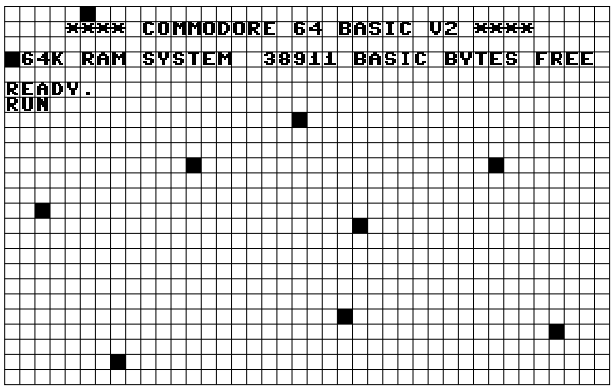


Figure 1: An intermediate position after a few iterations

// Linear-Feedback Shift Register (LFSR)

What is an LFSR in the first place?

An LFSR can be viewed as a generator that produces a repeating sequence of bits.

How does that help with generating random numbers, and what are "pseudorandom" numbers?

A true random number sequence—where the relationship between values is completely nondeterministic—is very difficult to achieve. A pseudorandom number generator is an approximation: it produces sequences that look random to an observer, even though the underlying process is completely deterministic.

How does that all fit together for filling our screen?

Well, let's simplify a bit and pretend our screen is only 2 rows of 8 columns.

Now let's say we fill the cells one after another. The number in each cell represents the order in which the cells are filled.

pseudo-code

```
...
fill cell_index
cell_index = cell_index+1
...
```

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

To get a more random feel, we could do `cell_index = cell_index + 2`.

First 8 iterations…

1		2		3		4	
5		6		7		8	

But then 9 would end up in the same place as 1.

9		2		3		4	
5		6		7		8	

Is there a number we could add to `cell_index` so that all cells get visited? Let's try `cell_index = cell_index + 3`

1			**2**			**3**	
	4			**5**			**6**

then …

1		**7**	2		**8**	3	
9	4		**10**	5		**11**	6

and finally

1	**12**	7	2	**13**	8	3	**14**
9	4	**15**	10	5	**16**	11	6

The problem is that if you do this (`cell_index + 3`) to fill the screen, that's not going to give you a random feel at all. You'll quickly recognize the pattern. You could try a bigger number like `cell_index + 7`, but that won't help much… the numbers are still too closely related.

Is there a way to visit all cells one after another, but in a less predictable way? Yes — and that's where the LFSR comes in.

LFSR explained

The LFSR is not addition-based; it uses ‘shift’ and ‘xor’ operations. Here’s how an 8-bit Galois LFSR works.

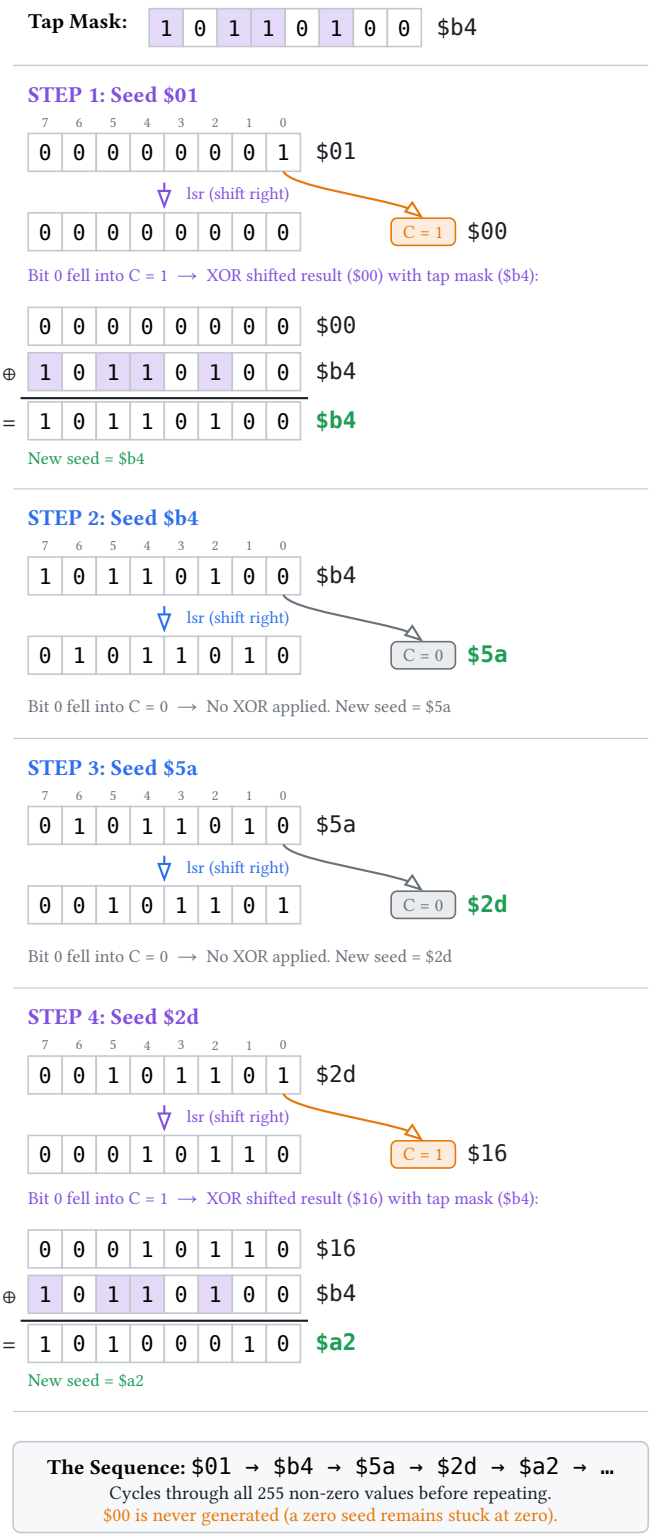


Figure 2: The 8-bit LFSR in Galois form

The tap mask $b4 may look like a magical value. This is actually a primitive polynomial. Without going too far into theory, a primitive polynomial guarantees that an 8-bit Galois LFSR runs through a maximal-length sequence.

Size	Maximal Value ($2^n - 1$)	Galois Hex Mask
8-bit	255	`$b4`
9-bit	511	`$110`
10-bit	1023	`$240`
11-bit	2047	`$500`
12-bit	4095	`$e08`
13-bit	8191	`$1c80`
14-bit	16383	`$3802`
15-bit	32767	`$6000`
16-bit	65535	`$b400`

Table 1: Galois LFSR tap masks for sizes 8 to 16 bits.

Although the sequence is still deterministic, filling the screen this way looks far more random, avoiding the repetitive patterns seen with simple addition.

Kick Assembler implementation for 8-bit and 10-bit sequences

```
next_lfsr_8bit: {
    lsr seed
    bcc no_feedback

    lda seed
    eor #$b4
    sta seed

no_feedback:
    rts
}

seed: .byte $01
```

```
next_lfsr_10bit:
    lsr seed+1
    ror seed
    bcc no_feedback

    lda seed
    eor #$40
    sta seed
    lda seed+1
    eor #$02
    sta seed+1
no_feedback:
    rts

seed: .word $0001
```

The 8-bit version provides numbers between 1 and 255, and the 10-bit version between 1 and 1023, both in pseudo-random order.

Note that for filling the screen, we will need numbers ranging from 0 to 999, so we will have to handle this in our code.

// Whole code for filling the screen using a 10-bit LFSR

```
.const OPCODE_JSR = $20
.const OPCODE_BIT = $2c

.pc = $02 "ZeroPage" virtual
seed:          .word 0
counter:       .word 0
target_ptr:    .word 0
save_target_ptr_hi:   .byte 0

BasicUpstart2(main)

main:
    jsr reset_lfsr_vars

forever:
    jsr wait
    // SMC - will become "bit fill" ($2c) when done.
    modify_jsr: jsr fill
    jmp forever

wait:
    bit $d011
    bpl *-3
    bit $d011
    bmi *-3
    rts

fill:
    // Testing that fill is no longer
    // called once done
    inc $d020
    ldx #8     // Fills 8 chars per frame
loop:
    jsr fill_one
    lda modify_jsr
    cmp #OPCODE_JSR
    bne stop
    dex
    bne loop
stop:
    rts

fill_one:
    jsr next_lfsr

    // Map (1 to 1023 -> 0 to 1022)
    sec
    lda seed
    sbc #1
    sta target_ptr
    lda seed+1
    sbc #0
    sta target_ptr+1
```

```
    // Out of the screen? (>= 1000)
    lda target_ptr+1
    cmp #>1000
    bcc valid
    bne fill_one
    lda target_ptr
    cmp #<1000
    bcs fill_one

valid:
   // Add base screen address ($0400)
    lda target_ptr+1
    sta save_target_ptr_hi
    clc
    adc #>$0400
    sta target_ptr+1

    ldy #0 // Offset not used

    lda #160 // Filled char
    sta (target_ptr), y

    lda save_target_ptr_hi
    clc
    adc #>$d800
    sta target_ptr+1
    lda #BLACK
    sta (target_ptr), y

    inc counter
    bne !+
    inc counter+1
!:
    lda counter
    cmp #<1000
    bne not_yet
    lda counter+1
    cmp #>1000
    bne not_yet

    // We are completely done filling
    // the screen here
    lda #0
    sta $d020
    lda #OPCODE_BIT
    sta modify_jsr      // Change jsr fill
                        // to bit fill

not_yet:
    rts

reset_lfsr_vars:
    lda #$01 // Whatever value here except for 0
    sta seed
```

```
    lda #$00
    sta seed+1
    sta counter
    sta counter+1
    rts

next_lfsr:
    lsr seed+1
    ror seed
    bcc no_feedback

    lda seed
    eor #$40
    sta seed
    lda seed+1
    eor #$02
    sta seed+1
no_feedback:
    rts
```

From a memory consumption point of view, this version is far more efficient than the precomputed random-number LUT we built in Round 4.

```
Memory Map
----------
Default-segment:
  *$0002-$0009 ZeroPage
  $0801-$080c Basic
  $080e-$08aa Basic End
```

> **Side note:** Should you always use LFSR instead of precomputed table for random numbers?
>
> **No.** It comes down to the trade-off between **execution speed** and **memory overhead**:
>
> - **Precomputed Lookup Table (LUT):** Faster to query (minimal CPU cycles), but consumes program memory or RAM. Ideal when you have free space and need low latency.
> - **Linear-Feedback Shift Register (LFSR):** Uses almost zero memory, but requires bitwise operations for every generated number. Ideal for memory-constrained hardware.

// Recap of the mnemonics we used so far

Mnemonic	Description
`lda`	Load Accumulator — Loads a value into register A.
`ldx`	Load X Register — Loads a value into register X.
`ldy`	Load Y Register — Loads a value into register Y.
`sta`	Store Accumulator — Writes the value from register A into memory.
`jmp`	Jump — Sets the Program Counter (PC) to jump directly to a target address.
`jsr`	Jump to Subroutine — Pushes the return address onto the stack and jumps to a subroutine.
`rts`	Return from Subroutine — Pulls the return address from the stack and resumes execution after the `jsr`.
`clc`	Clear Carry Flag — Clears the Carry flag ($C = 0$), required before 8-bit or 16-bit addition (`adc`).
`adc`	Add with Carry — Adds a value and the Carry flag to register A ($A = A + M + C$).
`bcc`	Branch if Carry Clear — Jumps to a target address if the Carry flag is clear ($C = 0$), e.g. when an addition produced no carry out of the byte (the result stayed ≤ 255).
`bne`	Branch if Not Equal — Jumps to a relative target address if the Zero flag is clear ($Z = 0$).
`dex`	Decrement X Register — Decreases register X by 1 ($X = X - 1$).
`dey`	Decrement Y Register — Decreases register Y by 1 ($Y = Y - 1$).
`inc`	Increment Memory — Increases a value in memory by 1 ($M = M + 1$).
`asl`	Arithmetic Shift Left. Shifts all bits in the accumulator or memory left by 1. Bit 7 is shifted into the Carry flag (C), and Bit 0 is filled with 0.
`lsr`	Logical Shift Right. Shifts all bits in the accumulator or memory right by 1. Bit 0 is shifted into the Carry flag (C), and Bit 7 is filled with 0.
`rol`	Rotate Left. Rotates all bits in the accumulator or memory left by 1 through the Carry flag. Bit 7 moves into Carry, and the old Carry value moves into Bit 0.
`ror`	Rotate Right. Rotates all bits in the accumulator or memory right by 1 through the Carry flag. Bit 0 moves into Carry, and the old Carry value moves into Bit 7.
`tya`	Transfer Y to Accumulator — Copies the value from register Y into register A ($A = Y$). Sets the Zero flag ($Z = 1$) if the value is 0 and the Negative flag ($N = 1$) if bit 7 is set.
`inx`	Increment X Register — Increases register X by 1 ($X = X + 1$).
`bit`	Bit Test. Performs a bitwise AND between the memory contents and the Accumulator without altering either register. Sets the Zero flag (Z) based on the AND result, and copies memory bit 7 into the Negative flag (N) and memory bit 6 into the Overflow flag (V).
`sec`	Set Carry Flag — Sets the Carry flag ($C = 1$), required before 8-bit or 16-bit subtraction (`sbc`).
`sbc`	Subtract with Carry — Subtracts a value and the borrow from register A ($A = A - M - (1 - C)$). Set the Carry with `sec` first so the initial borrow is 0.

New Mnemonics introduced in Round 5

Mnemonic	Description
`eor`	Exclusive OR — Performs a bitwise XOR between register A and memory ($A = A \oplus M$), storing the result in A. Sets the Zero flag ($Z = 1$) if the result is 0 and the Negative flag ($N = 1$) if bit 7 is set.

// Comments?

Something is wrong or missing? Feel free to send me a PM (OCA/Atlantis) on CSDb.

// What's coming next

- Start talking about flags and branches (all the 'B' mnemonics like beq, bne, bcc, and the like).