Bespoke

Very Fast^v Random

by Duck

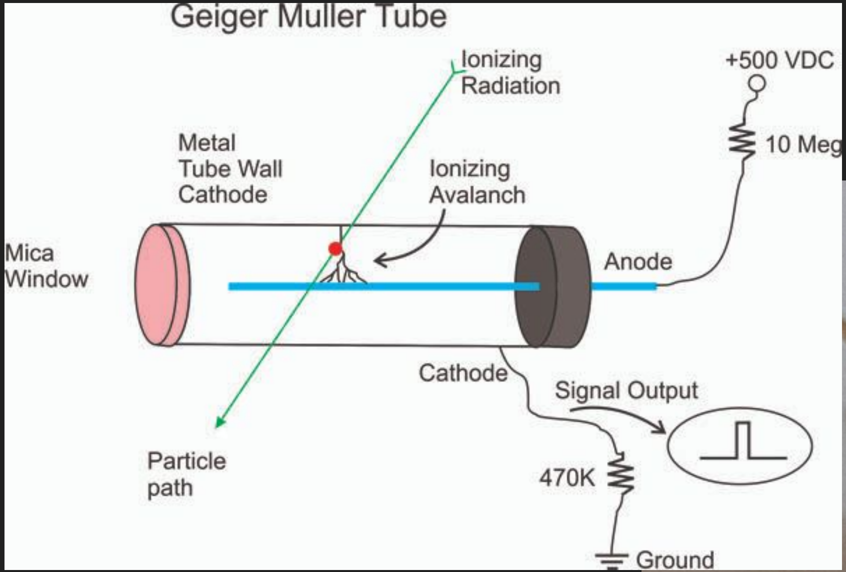
Actual End Goal

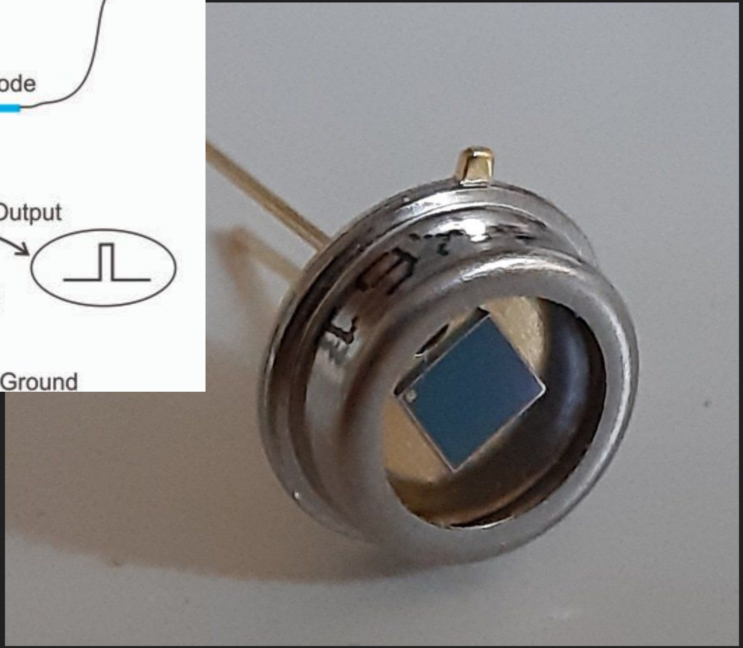
- Source of white noise for audio and filter analysis
- Spectrum from 0Hz to at least 200kHz
- Physically small
- Inexpensive
- Reliable
- (Long period if repeating)

Geiger–Müller or PIN Radiation Sensor

- True random

- Expensive
- Complicated
- ***Very*** low BW

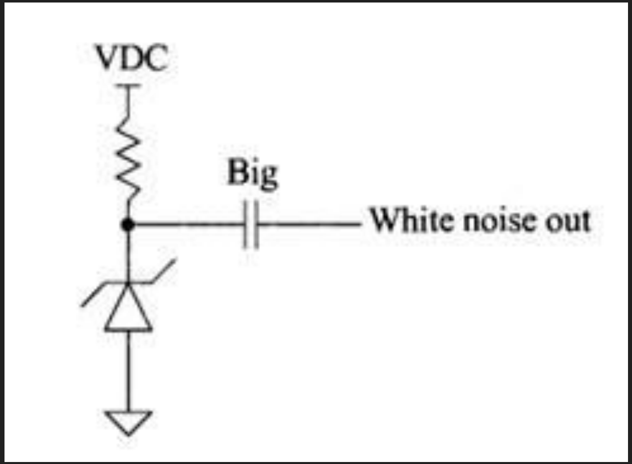


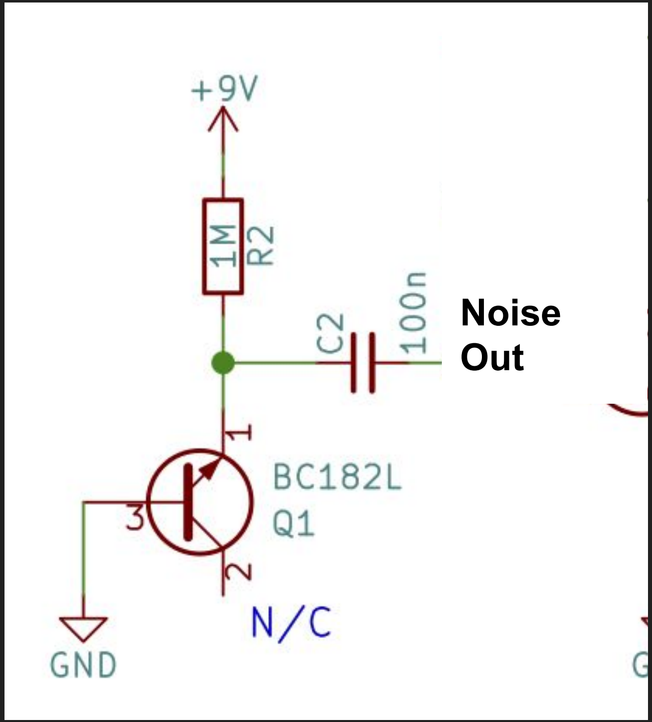


Zener/Avalanche/Flicker Noise

- Cheap
- True random

- Low amplitude
- Not guaranteed
  - Using parts outside of mfgr. specifications
- Changes during aging

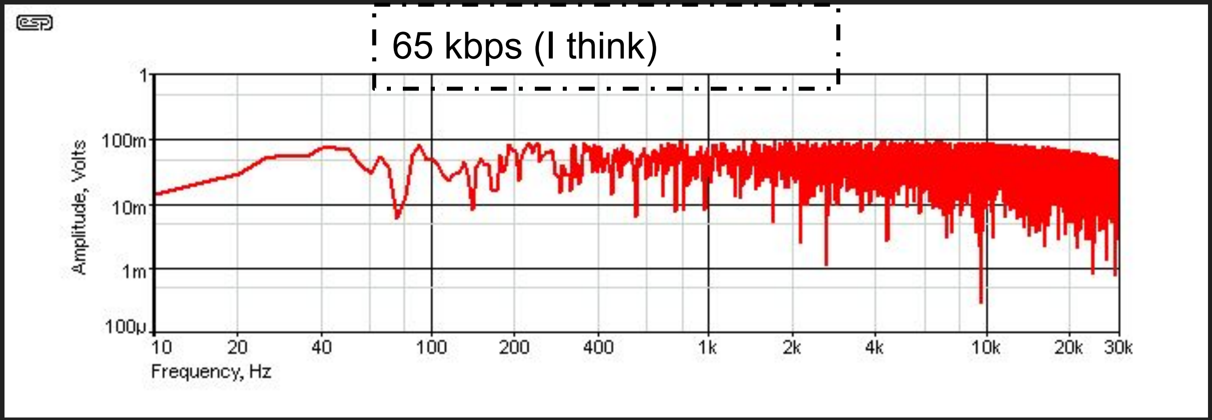




Going Digital

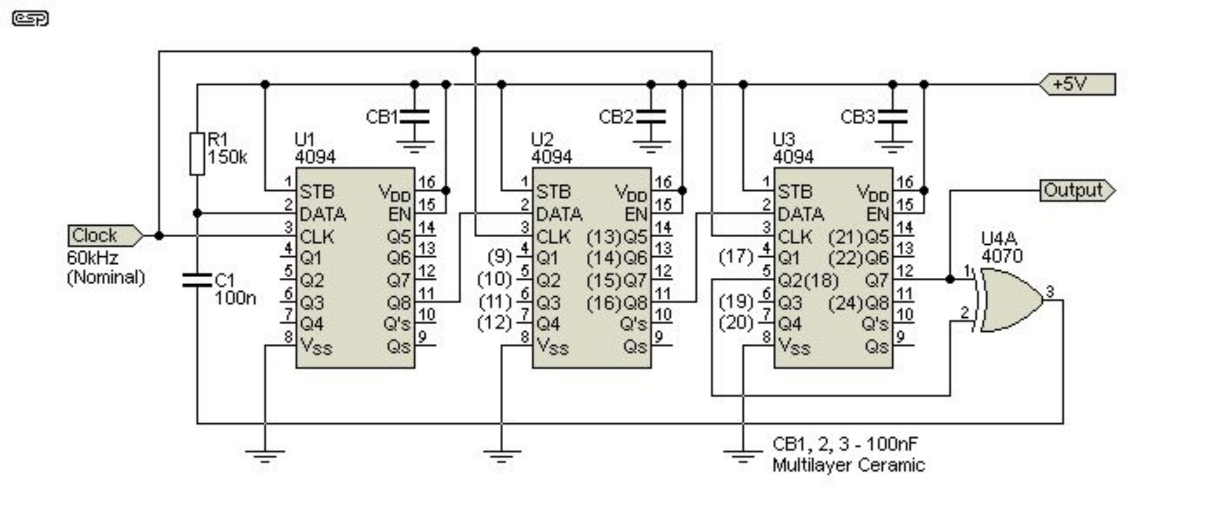
A random bit sequence at rate `f` bits per second approximates a white noise signal below about `f/3` Hz. [1] [2]

Pseudorandom bit sequences are fine too (if they have a long period)



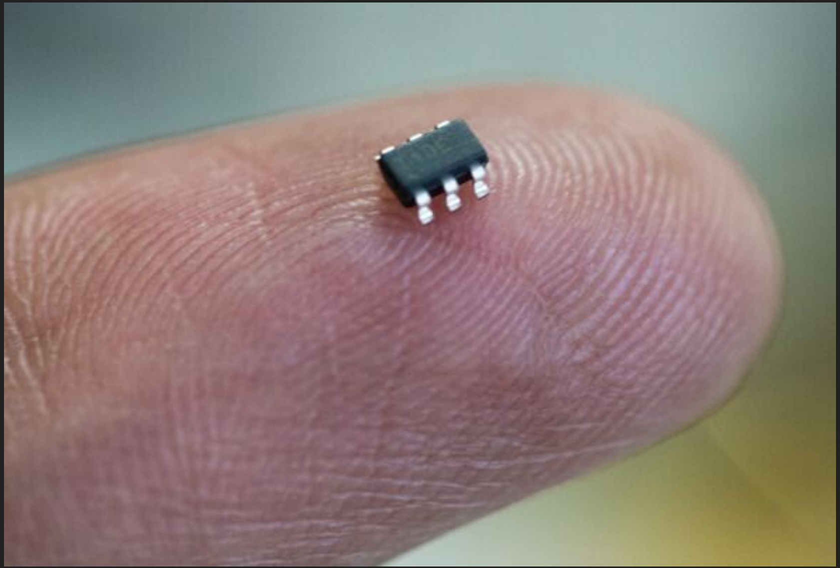
(Hardware) Linear Feedback Shift Registers (LFSR)

- Cheap
- Dependable
- Semi-complicated layout
- Large PCB footprint
- Need 4+ logic ICs
    - Even more for longer periods



Behold! The ATtiny10 from AVR/Microchip

- 8-bit architecture
- 1KB program flash
- 32B SRAM (B, not KB or MB)
- 12MHz max clock
- Direct hardware access
- 1 instruction per clock for most ALU instructions
- GCC supported
- $0.38 each (in quantity 1)
- (Most people would be better off with an Arduino)



Software LFSRs

- uC are cheap
- Small
- Dependable
- Too much math for high bit rates with a cheap/slow microcontroller

```c
# include <stdint.h>
unsigned lfsr1(void)
{
    uint16_t start_state = 0xACE1u;  /* Any nonzero start state will work. */
    uint16_t lfsr = start_state;
    uint16_t bit;                    /* Must be 16-bit to allow bit<<15 later in the code */
    unsigned period = 0;

    do
    {   /* taps: 16 14 13 11; feedback polynomial: x^16 + x^14 + x^13 + x^11 + 1 */
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) /* & 1u */;
        lfsr = (lfsr >> 1) | (bit << 15);
        ++period;
    }
    while (lfsr != start_state);

    return period;
}
```

Code from [3]

Fibonacci 32-bit LFSR

- (Quick guesstimates)
- Operations per bit of output:
  - 4x shift operations
  - 6x boolean operations
  - 1x output instruction
  - 1x unconditional jump
- 32-bit word size means most operations are now 4 clocks
  - So at least 42 instructions per bit of output
- 12MHz clock means only 100kHz noise

Confirming with GCC

```c
void lfsr_loop() {
    uint32_t bits = 0x1;
    uint32_t bit;
    while(1) {
        bit = ((bits >> 31) ^ (bits >> 21) ^ (bits >> 1) ^ bits) & 1;
        bits = (bits << 1) + bit;
        blah(bit);
    }
}
```

- Unoptimized 32-bit Fibonacci LFSR
- 184 instructions in loop
- Assuming 12 MIPS
- 21kHz of noise
- I want 10x that

```
00000024 <lfsr_loop>:
  24:   cf 93           push    r28
  26:   df 93           push    r29
  28:   cd b7           in      r28, 0x3d       ; 61
  2a:   de b7           in      r29, 0x3e       ; 62
  2c:   28 97           sbiw    r28, 0x08       ; 8
  2e:   0f b6           in      r0, 0x3f        ; 63
  30:   f8 94           cli
  32:   de bf           out     0x3e, r29       ; 62
  34:   0f be           out     0x3f, r0        ; 63
  36:   cd bf           out     0x3d, r28       ; 61
  38:   81 e0           ldi     r24, 0x01       ; 1
  3a:   90 e0           ldi     r25, 0x00       ; 0
  3c:   a0 e0           ldi     r26, 0x00       ; 0
  3e:   b0 e0           ldi     r27, 0x00       ; 0
  40:   89 83           std     Y+1, r24        ; 0x01
  42:   9a 83           std     Y+2, r25        ; 0x02
  44:   ab 83           std     Y+3, r26        ; 0x03
  46:   bc 83           std     Y+4, r27        ; 0x04
  48:   89 81           ldd     r24, Y+1        ; 0x01
  4a:   9a 81           ldd     r25, Y+2        ; 0x02
  4c:   ab 81           ldd     r26, Y+3        ; 0x03
  4e:   bc 81           ldd     r27, Y+4        ; 0x04
  50:   28 2f           mov     r18, r24
  52:   39 2f           mov     r19, r25
  54:   4a 2f           mov     r20, r26
  56:   5b 2f           mov     r21, r27
  58:   22 27           eor     r18, r18
  5a:   57 fd           sbrc    r21, 7
  5c:   23 95           inc     r18
  5e:   33 27           eor     r19, r19
  60:   44 27           eor     r20, r20
  62:   55 27           eor     r21, r21
  64:   89 81           ldd     r24, Y+1        ; 0x01
  66:   9a 81           ldd     r25, Y+2        ; 0x02
  68:   ab 81           ldd     r26, Y+3        ; 0x03
  6a:   bc 81           ldd     r27, Y+4        ; 0x04
  6c:   07 2e           mov     r0, r23
  6e:   75 e1           ldi     r23, 0x15       ; 21
  70:   b6 95           lsr     r27
  72:   a7 95           ror     r26
  74:   97 95           ror     r25
  76:   87 95           ror     r24
  78:   7a 95           dec     r23
  7a:   d1 f7           brne    .-12            ; 0x70 <__SREG__+0x31>
  7c:   70 2d           mov     r23, r0
  7e:   28 27           eor     r18, r24
  80:   39 27           eor     r19, r25
  82:   4a 27           eor     r20, r26
  84:   5b 27           eor     r21, r27
  86:   89 81           ldd     r24, Y+1        ; 0x01
  88:   9a 81           ldd     r25, Y+2        ; 0x02
  8a:   ab 81           ldd     r26, Y+3        ; 0x03
  8c:   bc 81           ldd     r27, Y+4        ; 0x04
  8e:   b6 95           lsr     r27
  90:   a7 95           ror     r26
  92:   97 95           ror     r25
  94:   87 95           ror     r24
  96:   28 27           eor     r18, r24
  98:   39 27           eor     r19, r25
  9a:   4a 27           eor     r20, r26
  9c:   5b 27           eor     r21, r27
  9e:   89 81           ldd     r24, Y+1        ; 0x01
  a0:   9a 81           ldd     r25, Y+2        ; 0x02
  a2:   ab 81           ldd     r26, Y+3        ; 0x03
  a4:   bc 81           ldd     r27, Y+4        ; 0x04
  a6:   82 27           eor     r24, r18
  a8:   93 27           eor     r25, r19
  aa:   a4 27           eor     r26, r20
  ac:   b5 27           eor     r27, r21
  ae:   81 70           andi    r24, 0x01       ; 1
  b0:   99 27           eor     r25, r25
  b2:   aa 27           eor     r26, r26
  b4:   bb 27           eor     r27, r27
  b6:   8d 83           std     Y+5, r24        ; 0x05
  b8:   9e 83           std     Y+6, r25        ; 0x06
  ba:   af 83           std     Y+7, r26        ; 0x07
  bc:   b8 87           std     Y+8, r27        ; 0x08
  be:   89 81           ldd     r24, Y+1        ; 0x01
  c0:   9a 81           ldd     r25, Y+2        ; 0x02
  c2:   ab 81           ldd     r26, Y+3        ; 0x03
  c4:   bc 81           ldd     r27, Y+4        ; 0x04
  c6:   28 2f           mov     r18, r24
  c8:   39 2f           mov     r19, r25
  ca:   4a 2f           mov     r20, r26
  cc:   5b 2f           mov     r21, r27
  ce:   22 0f           add     r18, r18
  d0:   33 1f           adc     r19, r19
  d2:   44 1f           adc     r20, r20
  d4:   55 1f           adc     r21, r21
  d6:   8d 81           ldd     r24, Y+5        ; 0x05
  d8:   9e 81           ldd     r25, Y+6        ; 0x06
  da:   af 81           ldd     r26, Y+7        ; 0x07
  dc:   b8 85           ldd     r27, Y+8        ; 0x08
  de:   82 0f           add     r24, r18
  e0:   93 1f           adc     r25, r19
  e2:   a4 1f           adc     r26, r20
  e4:   b5 1f           adc     r27, r21
  e6:   89 83           std     Y+1, r24        ; 0x01
  e8:   9a 83           std     Y+2, r25        ; 0x02
  ea:   ab 83           std     Y+3, r26        ; 0x03
  ec:   bc 83           std     Y+4, r27        ; 0x04
  ee:   8d 81           ldd     r24, Y+5        ; 0x05
  f0:   9e 81           ldd     r25, Y+6        ; 0x06
  f2:   af 81           ldd     r26, Y+7        ; 0x07
  f4:   b8 85           ldd     r27, Y+8        ; 0x08
  f6:   68 2f           mov     r22, r24
  f8:   79 2f           mov     r23, r25
  fa:   8a 2f           mov     r24, r26
  fc:   9b 2f           mov     r25, r27
  fe:   80 df           rcall   .-256           ; 0x0 <blah>
 100:   a3 cf           rjmp    .-186           ; 0x48 <__SREG__+0x9>
```

Galois LFSR

- Single shift
- Single conditional XOR
- Still dealing with 32-bit words

(Code is modified from [3] and is only 16-bit))

```c
void galois_lfsr_16(void)
{
    const uint16_t polynomial = 0x002D;
    uint16_t state = 1;
    while (1) {
        uint16_t msb = state & 0x8000;
        state <<= 1;
        if (msb) {
            state ^= polynomial;
        }
        blah(msb);
    }
}
```

Look at that Polynomial

0x002D

- Only has set bits in the low byte
- In `state ^= polynomial;`, only the low byte of \`state\` is affected

- Can we find a 32-bit polynomial with ones *only* in the low byte?
- It would make this faster on an 8-bit architecture

Brute Force Search

- An n-bit polynomial is “maximal” if it takes 2^n-1 cycles for the state to repeat
- For each candidate polynomial
  - Initialize the LFSR with `uint32_t state = 1;`
  - Step the LFSR until the current state equals the first state
  - Or until we hit 2^n-1 cycles
- Naively, only 255 possibly polynomials
  - Something something about alway a 1 in the lower bit??? So only 127?
- 2^{32}-1 == 4294967295 is a big number but not *that* big

24-bit Brute Force

```
duck@alura:~/Projects/fast_lfsr$ time ./test
poly: (27) 0x1b
poly: (135) 0x87
poly: (177) 0xb1
poly: (219) 0xdb
poly: (245) 0xf5

real	0m2.896s
user	0m2.895s
sys	0m0.001s
```

32-bit Brute Force

- Found 0xaf, 0xc5, 0xf5
- Took 12 minutes

40-bit Brute Force

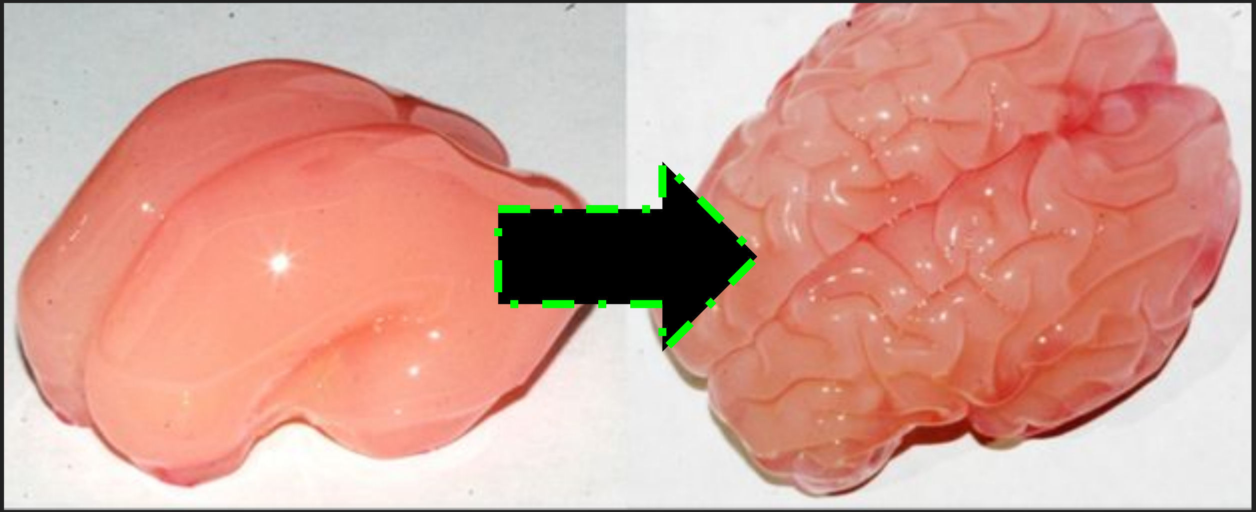
- Multi-threaded across a 48-core Xeon E5-2678 v3 @ 2.50GHz

```
  1  [|||||||||||||||||||||||||||||||||100.0%]   13 [|||||||||||||||||||||||||||||||||100.0%]   25 [|||||||||||||||||||||||||||||||||100.0%]   37 [|||||||||||||||||||||||||||||||||100.0%]
  2  [|||||||||||||||||||||||||||||||||100.0%]   14 [|||||||||||||||||||||||||||||||||100.0%]   26 [|||||||||||||||||||||||||||||||||100.0%]   38 [|||||||||||||||||||||||||||||||||100.0%]
  3  [|||||||||||||||||||||||||||||||||100.0%]   15 [|||||||||||||||||||||||||||||||||100.0%]   27 [|||||||||||||||||||||||||||||||||100.0%]   39 [|||||||||||||||||||||||||||||||||100.0%]
  4  [|||||||||||||||||||||||||||||||||100.0%]   16 [|||||||||||||||||||||||||||||||||100.0%]   28 [|||||||||||||||||||||||||||||||||100.0%]   40 [|||||||||||||||||||||||||||||||||100.0%]
  5  [|||||||||||||||||||||||||||||||||100.0%]   17 [|||||||||||||||||||||||||||||||||100.0%]   29 [|||||||||||||||||||||||||||||||||100.0%]   41 [|||||||||||||||||||||||||||||||||100.0%]
  6  [|||||||||||||||||||||||||||||||||100.0%]   18 [|||||||||||||||||||||||||||||||||100.0%]   30 [|||||||||||||||||||||||||||||||||100.0%]   42 [|||||||||||||||||||||||||||||||||100.0%]
  7  [|||||||||||||||||||||||||||||||||100.0%]   19 [|||||||||||||||||||||||||||||||||100.0%]   31 [|||||||||||||||||||||||||||||||||100.0%]   43 [|||||||||||||||||||||||||||||||||100.0%]
  8  [|||||||||||||||||||||||||||||||||100.0%]   20 [|||||||||||||||||||||||||||||||||100.0%]   32 [|||||||||||||||||||||||||||||||||100.0%]   44 [|||||||||||||||||||||||||||||||||100.0%]
  9  [|||||||||||||||||||||||||||||||||100.0%]   21 [|||||||||||||||||||||||||||||||||100.0%]   33 [|||||||||||||||||||||||||||||||||100.0%]   45 [|||||||||||||||||||||||||||||||||100.0%]
  10 [|||||||||||||||||||||||||||||||||100.0%]   22 [|||||||||||||||||||||||||||||||||100.0%]   34 [|||||||||||||||||||||||||||||||||100.0%]   46 [|||||||||||||||||||||||||||||||||100.0%]
  11 [|||||||||||||||||||||||||||||||||100.0%]   23 [|||||||||||||||||||||||||||||||||100.0%]   35 [|||||||||||||||||||||||||||||||||100.0%]   47 [|||||||||||||||||||||||||||||||||100.0%]
  12 [|||||||||||||||||||||||||||||||||100.0%]   24 [|||||||||||||||||||||||||||||||||100.0%]   36 [|||||||||||||||||||||||||||||||||100.0%]   48 [|||||||||||||||||||||||||||||||||100.0%]
  Mem[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||      10.8G/126G]   Tasks: 64, 216 thr; 48 running
  Swp[||                                                                          21.2M/976M]   Load average: 36.31 11.88 4.20
                                                                                                 Uptime: 191 days(!), 01:32:59

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 6690 duck       20   0  518M  1840  1568 S 4796  0.0  1h05:46 ./test
 6731 duck       20   0  518M  1840  1568 R 101.  0.0  1:22.29 ./test
 6693 duck       20   0  518M  1840  1568 R 100.  0.0  1:21.96 ./test
 6700 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.30 ./test
 6691 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.28 ./test
 6701 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.31 ./test
 6703 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.16 ./test
 6702 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.30 ./test
 6736 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.08 ./test
 6711 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.31 ./test
 6705 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.31 ./test
 6718 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.31 ./test
 6698 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.30 ./test
 6709 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.19 ./test
 6717 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.30 ./test
 6712 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.30 ./test
 6695 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.07 ./test
 6729 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.31 ./test
 6735 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.31 ./test
 6714 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.28 ./test
 6721 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.20 ./test
 6720 duck       20   0  518M  1840  1568 R 100.  0.0  1:22.28 ./test
 6704 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6706 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.26 ./test
 6707 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6716 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6725 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6722 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6728 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6730 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6696 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6713 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6699 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6723 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.29 ./test
 6694 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.27 ./test
 6727 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6738 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.30 ./test
 6732 duck       20   0  518M  1840  1568 R 99.8  0.0  1:22.24 ./test
```

48-bit Brute Force

- Killed it after 2 weeks
- Needed a couple more months to finish



Academic Code and Stack Exchange to the Rescue

- Method for checking if a polynomial is maximal [4]
  - Requires factoring 2^n-1 and fast-forwarding the LFSR
- Some more explanations in [5]
- I don't really know what I'm doing
- Code to fast-forward LFSRs in [6]
  - Academic-style code 😠
  - Took a couple hours to figure out how to use
- Can find all lower-byte-only full-byte maximal polynomials up to 256-bit in
- 51 minutes (single threaded)

Results

```
16: maximals = [45, 57, 63, 83, 189, 215]
24: maximals = [27, 135, 177, 219, 245]
32: maximals = [175, 197, 245]
40: maximals = [57, 215]
48: maximals = [183]
56: maximals = [149]
64: maximals = [27, 29, 245]
72: maximals = [95]
80: maximals = [175]
88: maximals = []
96: maximals = [221]
104: maximals = []
112: maximals = []
120: maximals = [231]
128: maximals = [135]
136: maximals = []
```

```
144: maximals = [149]
152: maximals = [77]
160: maximals = [45, 57]
168: maximals = []
176: maximals = [189]
184: maximals = []
192: maximals = []
200: maximals = [45]
208: maximals = []
216: maximals = [139, 189]
224: maximals = []
232: maximals = []
240: maximals = []
248: maximals = []
256: maximals = []
```

Implementation

- C is too high level
- Luckily AVR8 assembly is moderately easy
- One more constraint: I need a constant time per bit of output

Paraphrased Assembly

```
load_immediate r19, 0xaf
load_immediate r20, 1
load_immediate r21, 1
load_immediate r22, 1
load_immediate r23, 1
myloop:                                               |  Clock cycles:
    logical_shift_left          r20                   |  1
    rotate_left_through_carry r21                     |  1
    rotate_left_through_carry r22                     |  1
    rotate_left_through_carry r23                     |  1
    branch_if_carry_cleared next_instruction          |  1 or 2
        xor r20, r19                                  |  1 (or 0 when skipped)
    next_instruction:
    out  OUTPUT_PORT, r23                             |  1
    rjump myloop                                      |  2
```

Results

- 1.3 Mbps of PRNG
- 6 minute period (time between repeats)
- Good for 440kHz of white noise
- 3mm x 3mm (plus passives) of PCB footprint
- $0.38

References

[1] https://sound-au.com/project182.htm White noise and Pink noise generation

[2] https://www3.advantest.com/documents/11348/3e95df23-22f5-441e-8598-f1d99c2382cb PRNG and spectrum stuff

[3] https://en.wikipedia.org/wiki/Linear-feedback_shift_register

[4] https://crypto.stackexchange.com/a/12835 Finding a maximal length polynomial

[5] https://mathoverflow.net/a/46983 More maximal polynomial math

[6] https://github.com/markagold1/LFSR-LAB Academic code for working with LFSRs

https://ww1.microchip.com/downloads/en/DeviceDoc/ATtiny4-5-9-10-Data-Sheet-DS40002060A.pdf ATtiny10 datasheet

https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-Instruction-Set-Manual-DS40002198A.pdf AVR8 Instruction Set