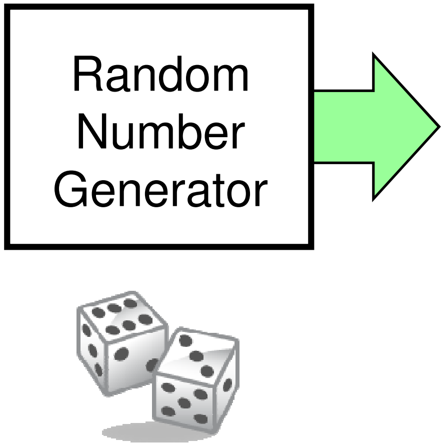
ECE 4514
Digital Design II
Spring 2008

Lecture 6:
A Random Number Generator in Verilog

A *Design* Lecture

Patrick Schaumont

What is a random number generator?

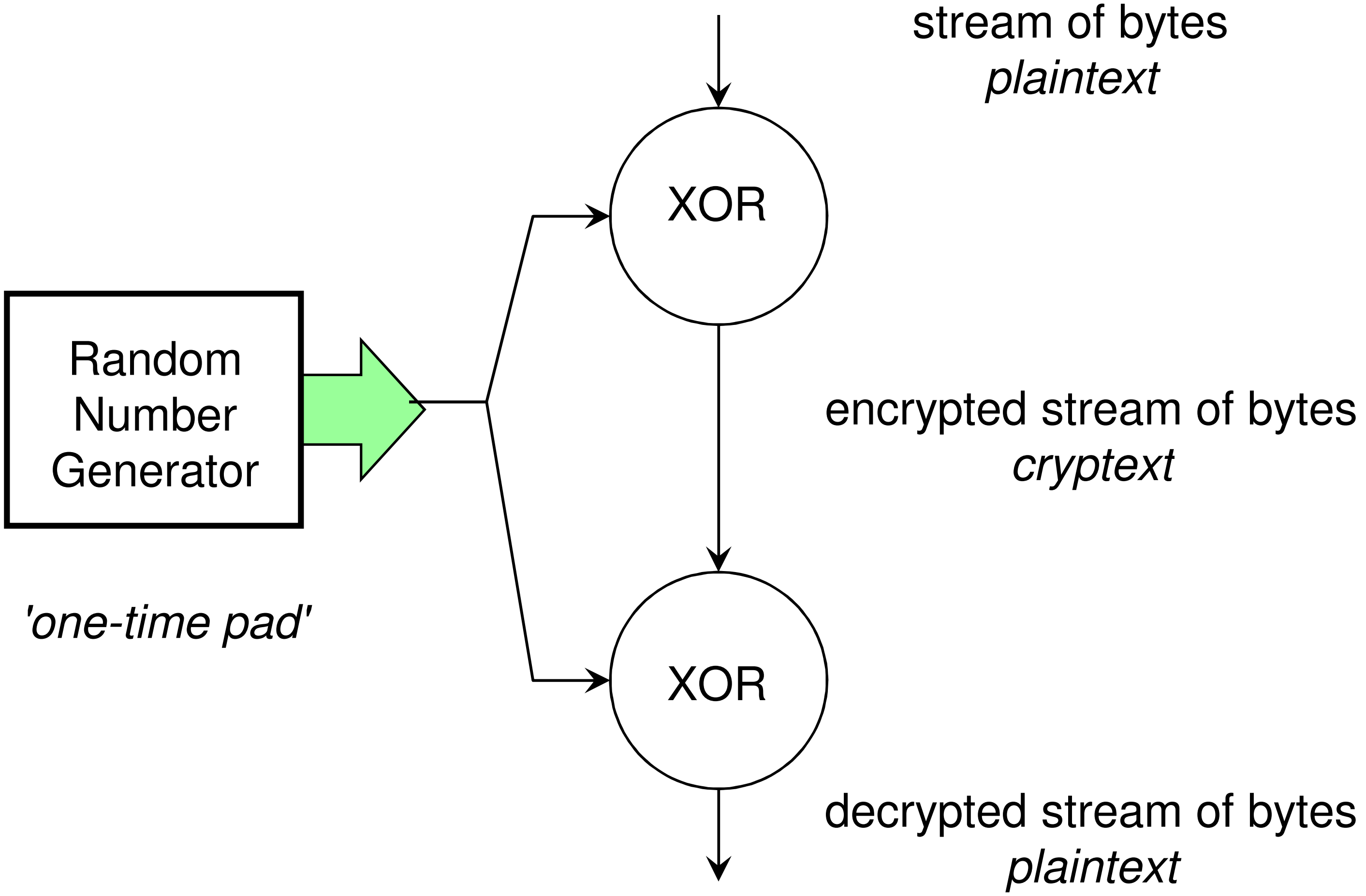


11, 86, 82, 52, 60, 46, 64, 10, 98, 2, ...

What do I do with randomness?

- ❑ Play games!
  - Have the monsters appear in different rooms every time
- ❑ Do statistical simulations
  - Simulate customers in a shopping center (find the best spot for a new Chuck E Cheese)
- ❑ Run security protocols
  - Make protocol resistent against replay
- ❑ Encrypt documents
  - Use random numbers as key stream

Encrypt Documents



stream of bytes

plaintext

Random Number Generator

XOR

encrypted stream of bytes

cryptext

XOR

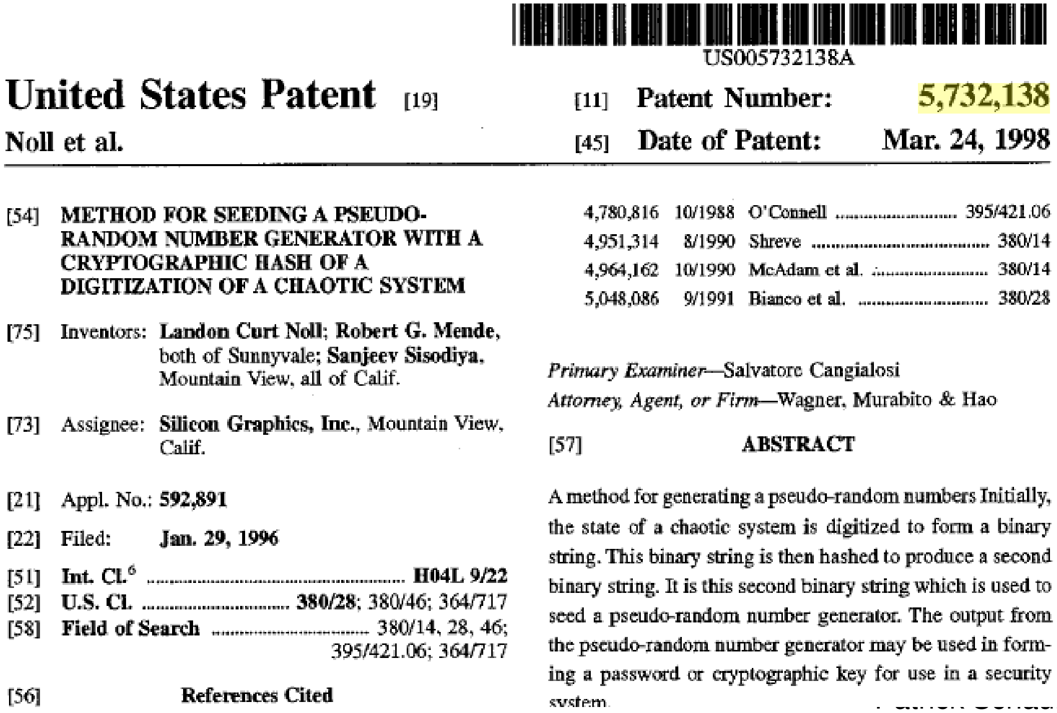
decrypted stream of bytes

plaintext

'one-time pad'

Random numbers by physical methods

- Use dice, coin flips, roulette
- Use thermal noise (diodes and resistors)
- Use clock jitter (use ring oscillators)
- Use radioactive decay
- Use Lava Lamps
  - Patented!



US005732138A

United States Patent [19]

Noll et al.

[11] **Patent Number:** 5,732,138

[45] **Date of Patent:** **Mar. 24, 1998**

[54] **METHOD FOR SEEDING A PSEUDO-RANDOM NUMBER GENERATOR WITH A CRYPTOGRAPHIC HASH OF A DIGITIZATION OF A CHAOTIC SYSTEM**

[75] Inventors: **Landon Curt Noll**; **Robert G. Mende**, both of Sunnyvale; **Sanjeev Sisodiya**, Mountain View, all of Calif.

[73] Assignee: **Silicon Graphics, Inc.**, Mountain View, Calif.

[21] Appl. No.: **592,891**

[22] Filed: **Jan. 29, 1996**

[51] **Int. Cl.**^{6} **H04L 9/22**
[52] **U.S. Cl.** **380/28**; 380/46; 364/717
[58] **Field of Search** 380/14, 28, 46; 395/421.06; 364/717

[56] **References Cited**

4,780,816	10/1988	O'Connell	395/421.06
4,951,314	8/1990	Shreve	380/14
4,964,162	10/1990	McAdam et al.	380/14
5,048,086	9/1991	Bianco et al.	380/28

Primary Examiner—Salvatore Cangialosi
Attorney, Agent, or Firm—Wagner, Murabito & Hao

[57] **ABSTRACT**

A method for generating a pseudo-random numbers Initially, the state of a chaotic system is digitized to form a binary string. This binary string is then hashed to produce a second binary string. It is this second binary string which is used to seed a pseudo-random number generator. The output from the pseudo-random number generator may be used in forming a password or cryptographic key for use in a security system.

Random numbers by computational methods

- ❑ Not truly random, but *pseudo* random
  - meaning, after some time the same sequence returns
- ❑ Linear Congruential Generator

$$x(n+1) = [\ a.x(b) + b\ ]\ \mathrm{mod}\ m$$

Eg. a = 15, b = 5, m = 7

a, b, m must be chosen carefully!
for a maximum lenth sequence

X(0) = 1
X(1) = (15 + 5) mod 7 = 6
x(2) = (15*6 + 5) mod 7 = 4
x(3) = 2
x(4) = 0
x(5) = 5
x(6) = 3
x(7) = 1
x(8) = ...

A quick way to generate random numbers

- Verilog has a buildin random number generator

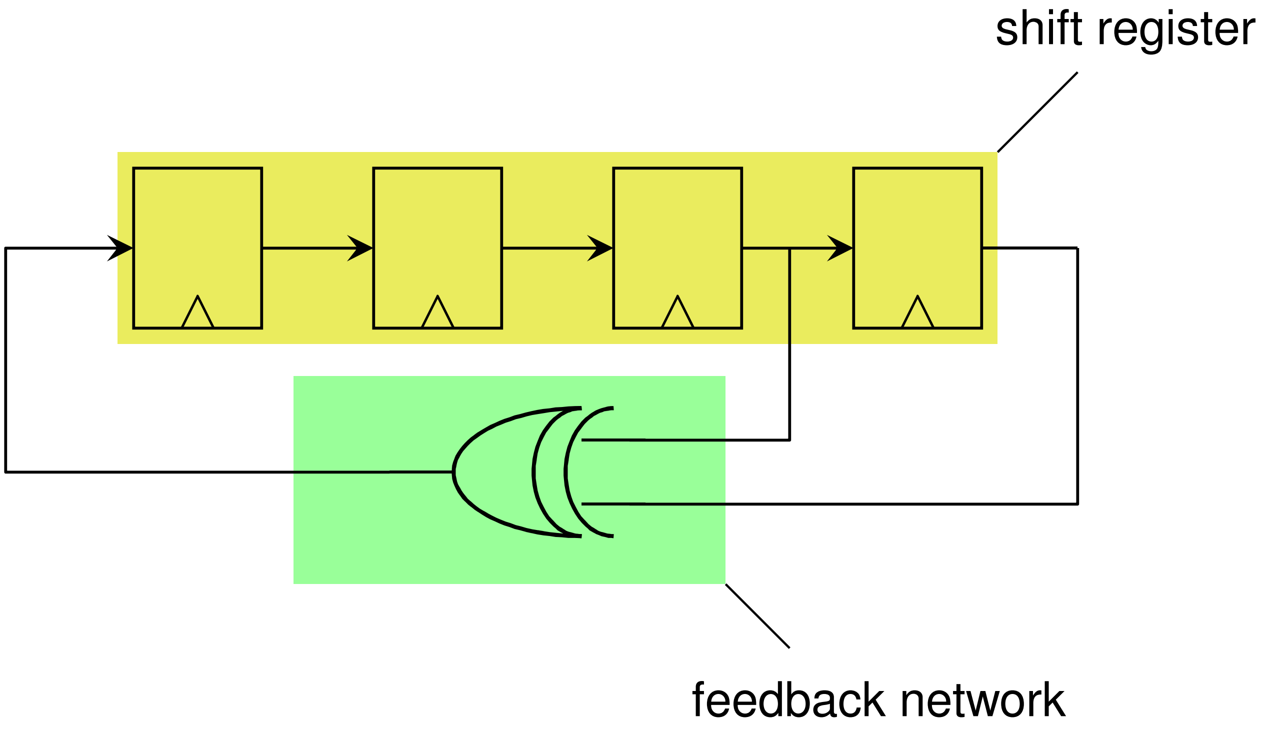
```
module random(q);
  output [0:31] q;
  reg [0:31] q;

  initial
    r_seed = 2;

  always
    #10 q = $random(r_seed);
endmodule
```

- Nice, but only for testbenches …

- Instead, we want an hardware implementation

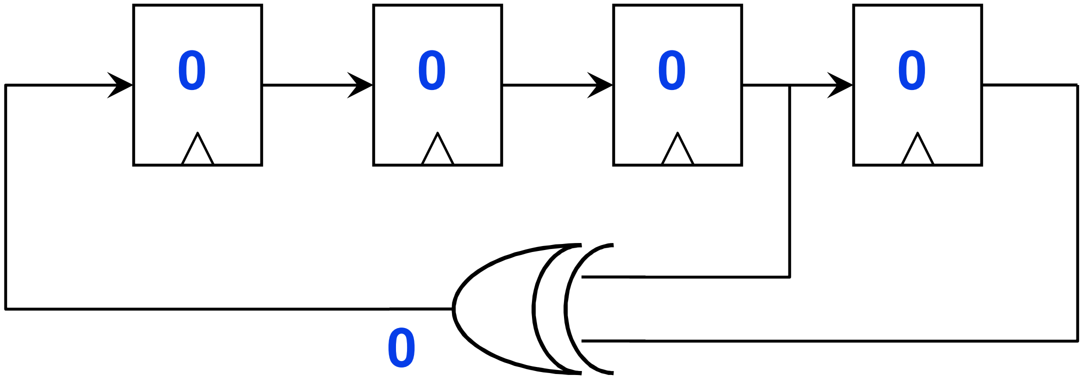
Linear Feedback Shift Register

- Pseudo Random Numbers in Digital Hardware



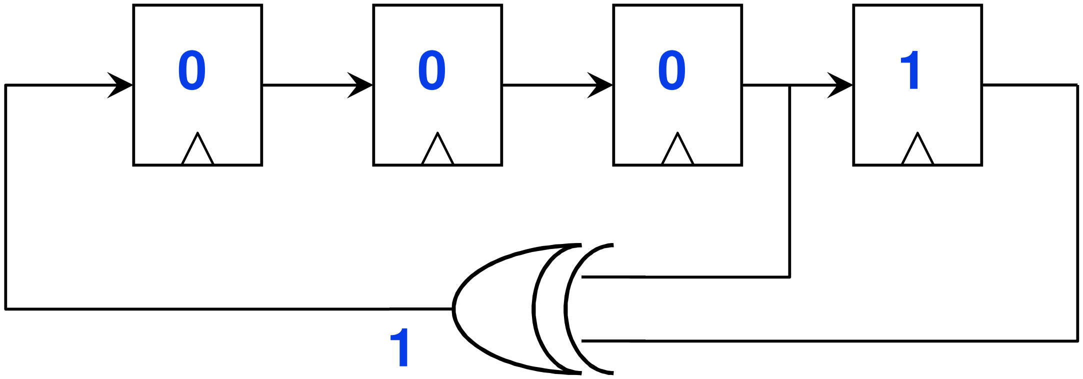
Linear Feedback Shift Register

- All zeroes
  - not very useful ...



Linear Feedback Shift Register

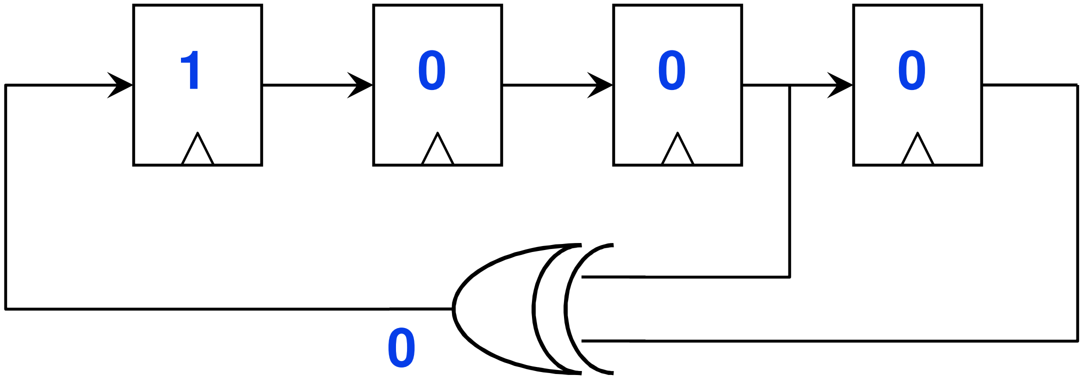
- Non-zero state is more interesting



1

Linear Feedback Shift Register

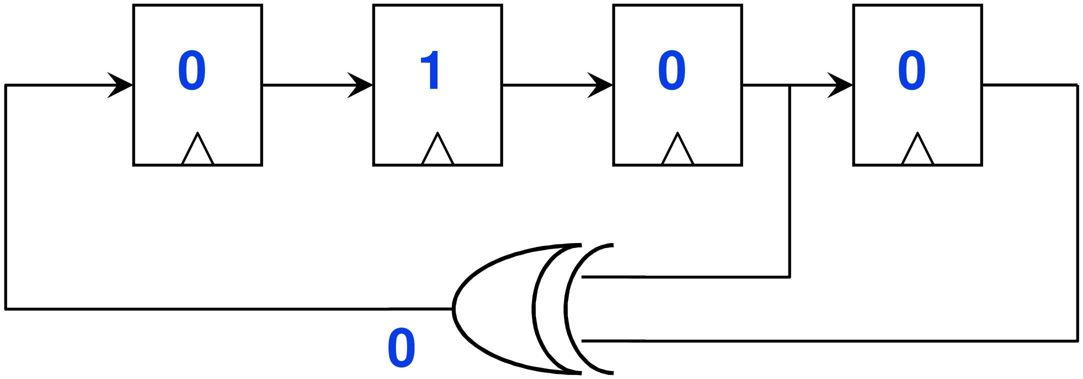
- Non-zero state is more interesting



10

Linear Feedback Shift Register

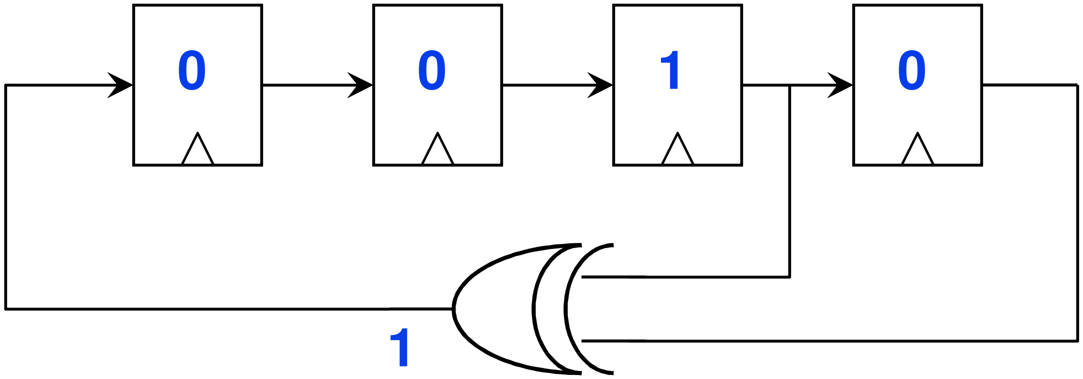
- Non-zero state is more interesting



100

Linear Feedback Shift Register

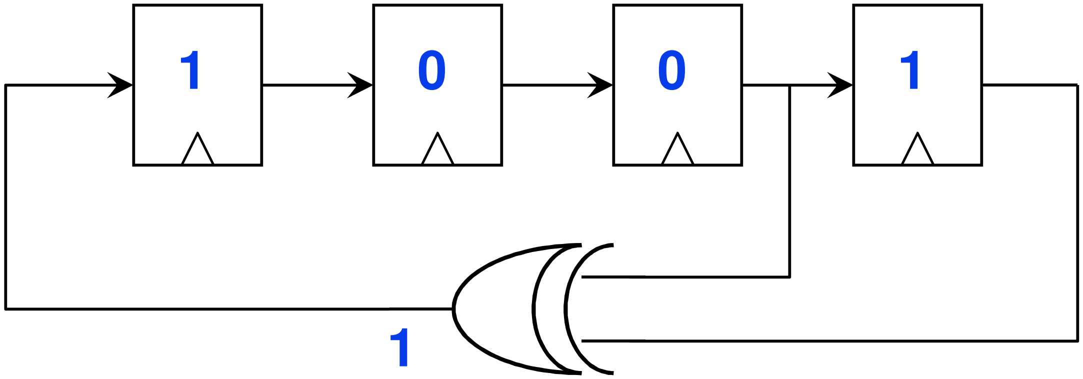
- Non-zero state is more interesting



1001

Linear Feedback Shift Register

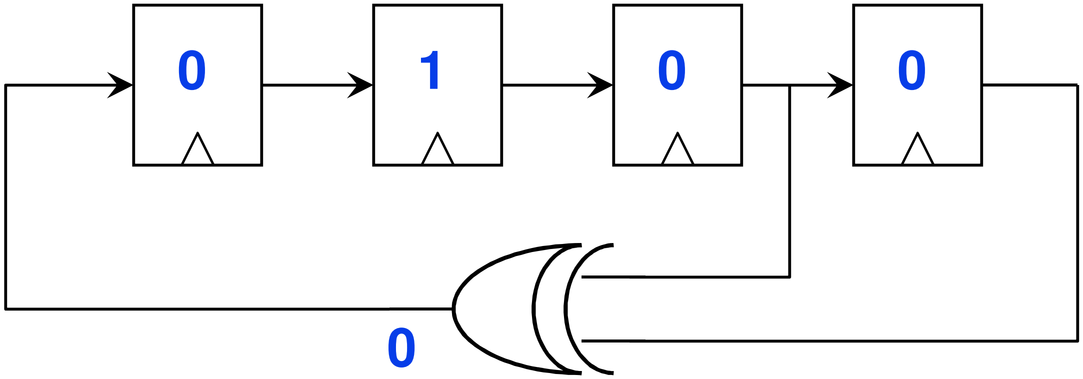
- Non-zero state is more interesting



10011

Linear Feedback Shift Register

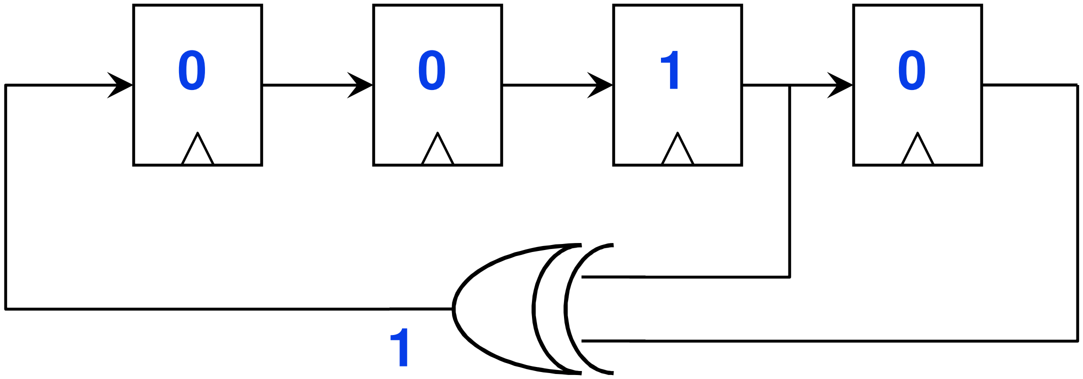
- Non-zero state is more interesting



100110

Linear Feedback Shift Register

- Non-zero state is more interesting

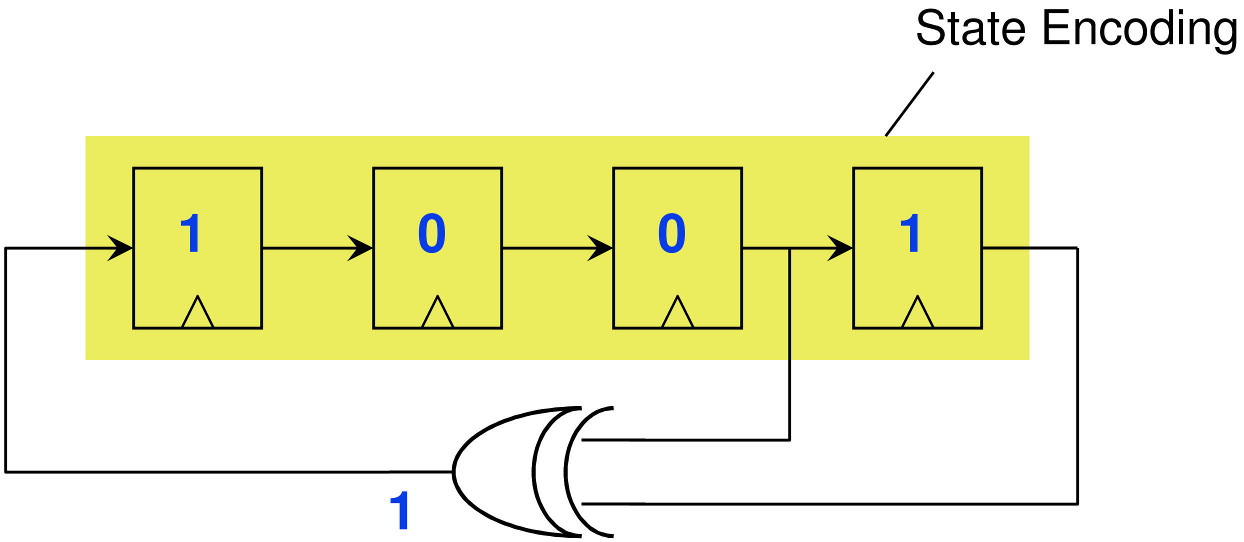


1001101

etc ...

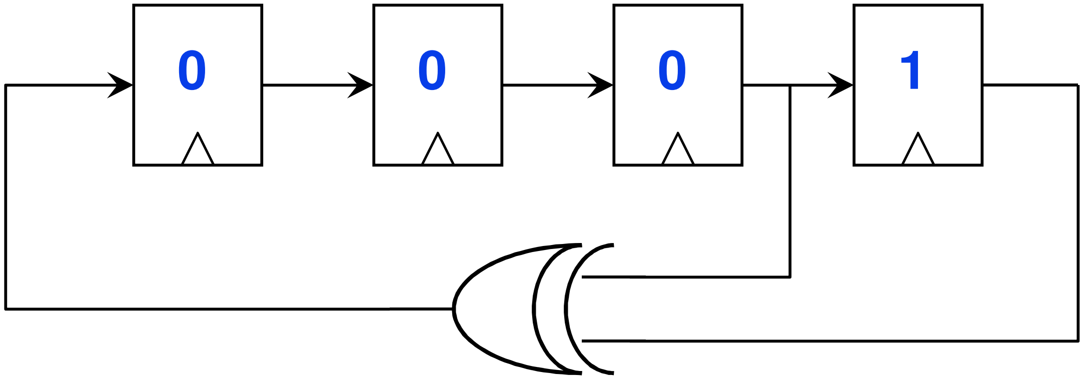
Linear Feedback Shift Register

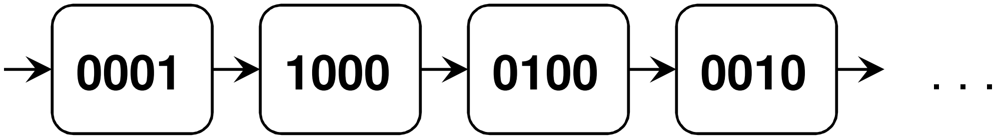
- This is actually a finite state machine



Linear Feedback Shift Register

- ❑ This is actually a finite state machine



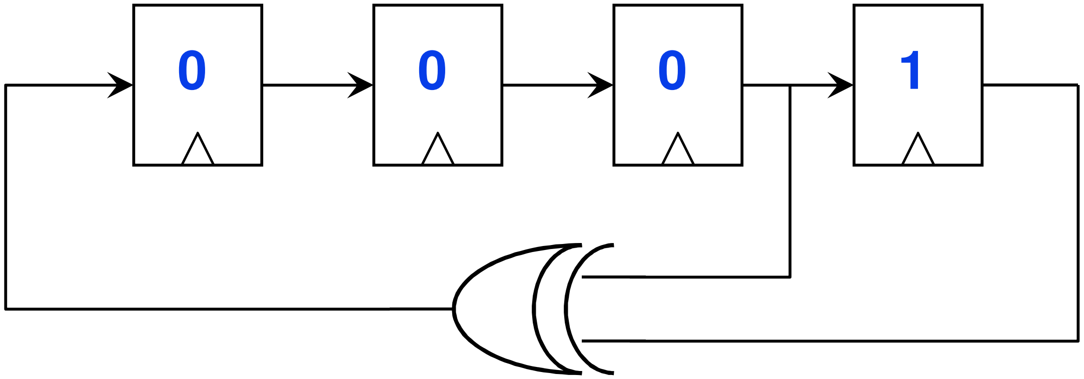


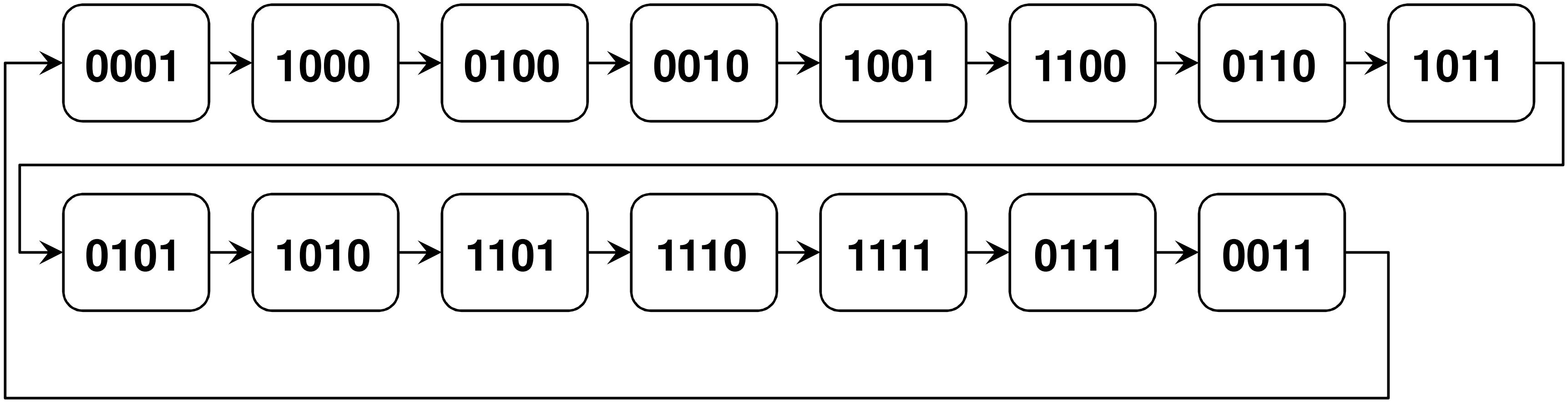
0001 → 1000 → 0100 → 0010 → . . .

How many states will you see?

Linear Feedback Shift Register

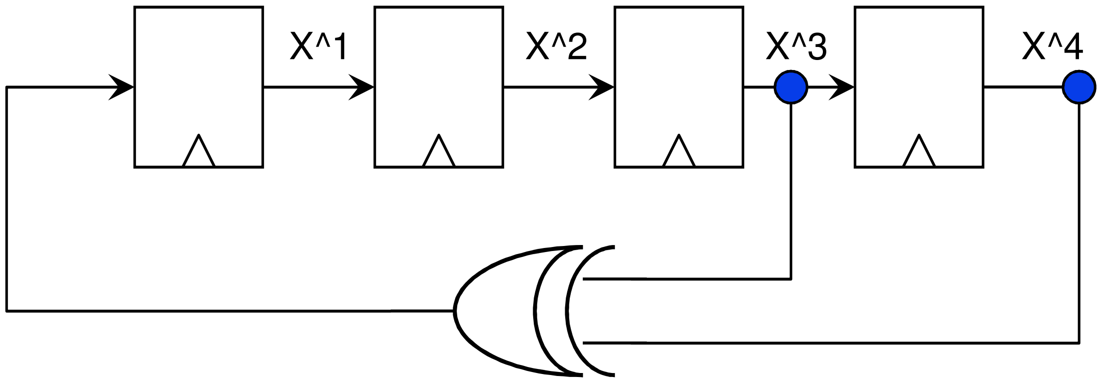
- 15 states





Linear Feedback Shift Register

- ❑ We can specify an LFSR by means of the *characteristic polynomial* (also called feedback polynomial)



$$P(x) = x^4 + x^3 + 1$$

There exists elaborate finite-field math to analyze the properties of an LFSR - outside of the scope of this class

Linear Feedback Shift Register

- So, knowing the polynomial you can also draw the LFSR

$$P(x) = x^8 + x^6 + x^5 + x^4 + 1$$

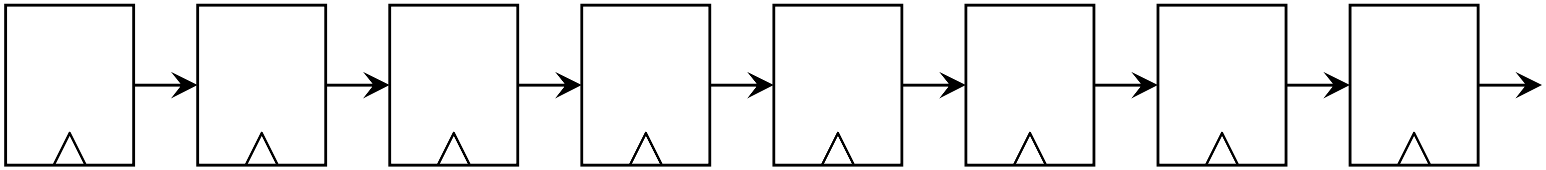
How many taps ?
How many 2-input XOR?

Linear Feedback Shift Register

- So, knowing the polynomial you can also draw the LFSR

$$P(x) = x^8 + x^6 + x^5 + x^4 + 1$$

How many taps ? 8
How many 2-input XOR?

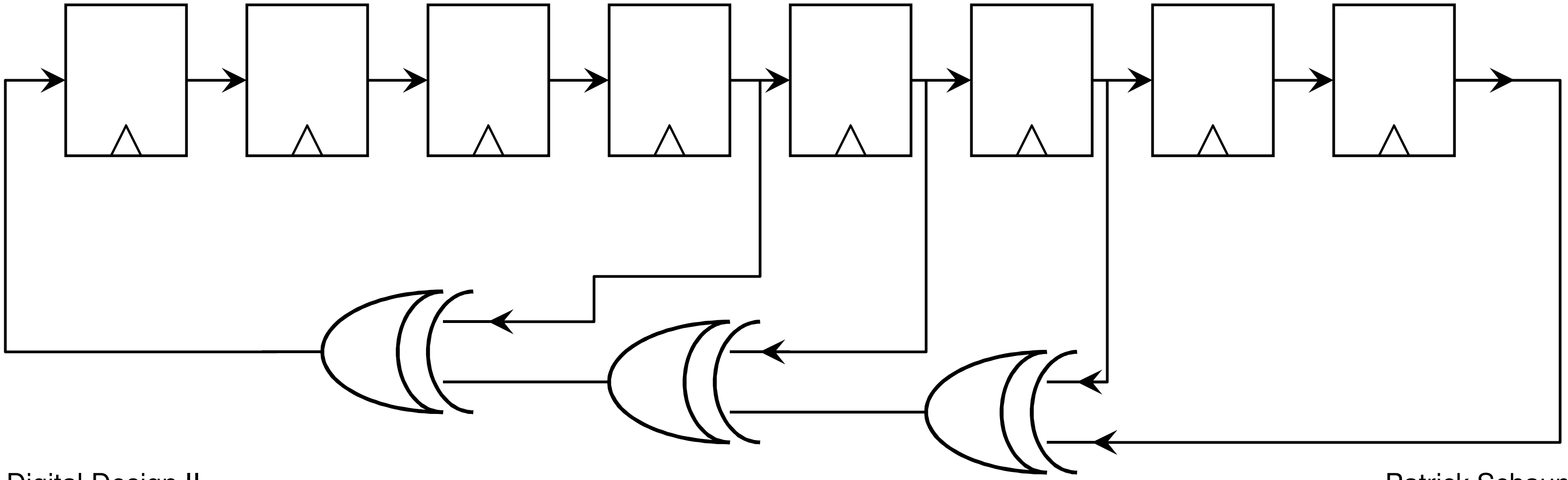


Linear Feedback Shift Register

- ❑ So, knowing the polynomial you can also draw the LFSR

$$P(x) = x^8 + x^6 + x^5 + x^4 + 1$$

How many taps ? 8
How many 2-input XOR? 3



Linear Feedback Shift Register

- Certain polynomials generate very long state sequences. These are called maximal-length LFSR.

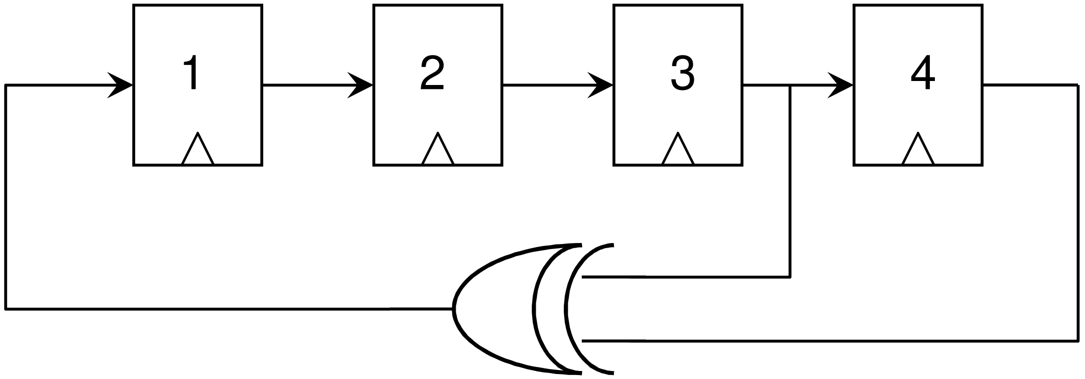
P(X) = x^153 + x^152 + 1

is a maximum-length feedback polynomial

State machine with 2 ^ 153 -1 states ..

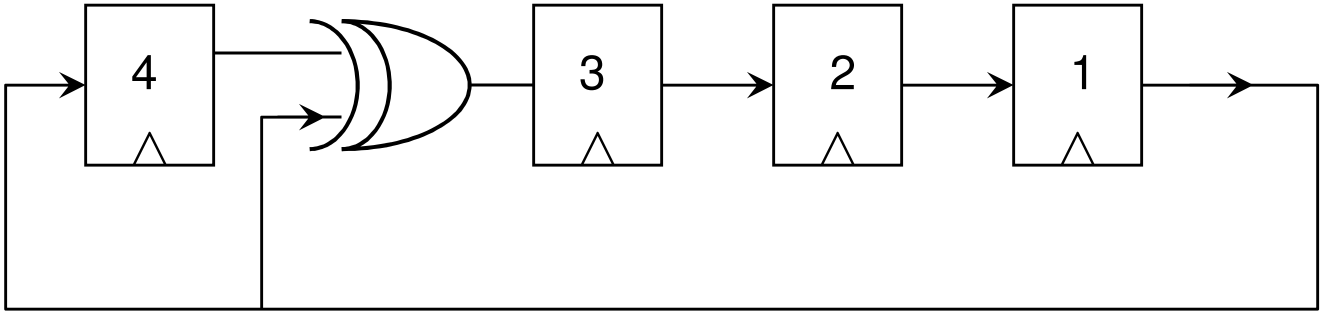
Fibonacci and Galois LFSR

- This format is called a Fibonacci LFSR



Fibonacci
~1175-1250

- Can be converted to an equivalent Galois LFSR

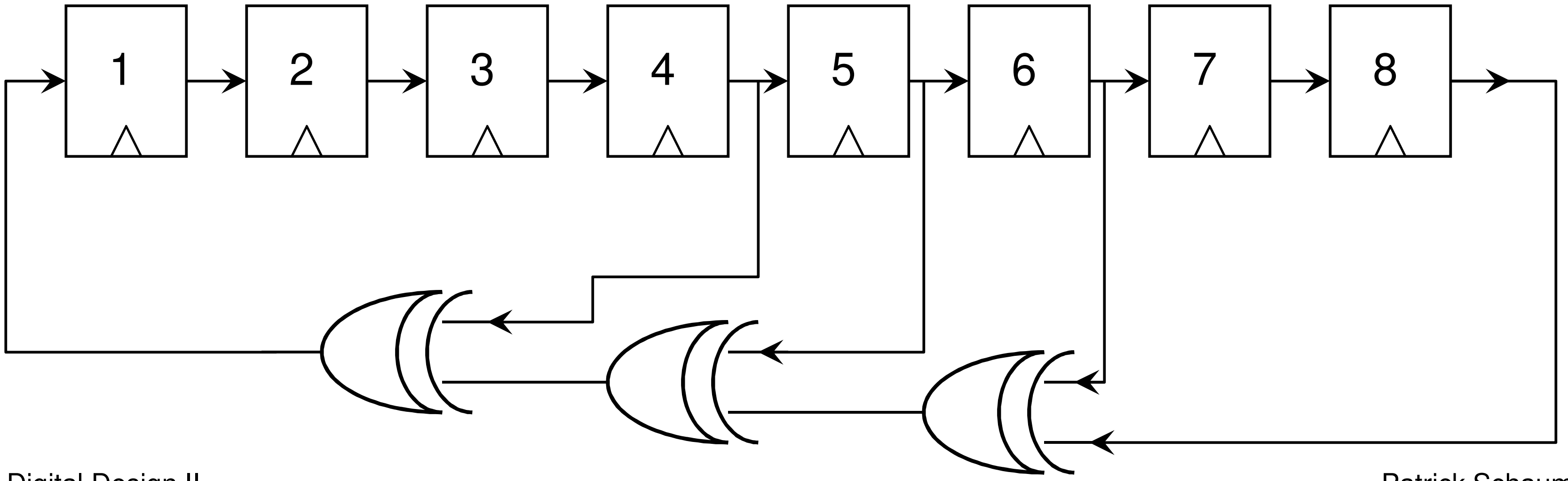


Evariste
Galois
1811-1832

Fibonacci and Galois LFSR

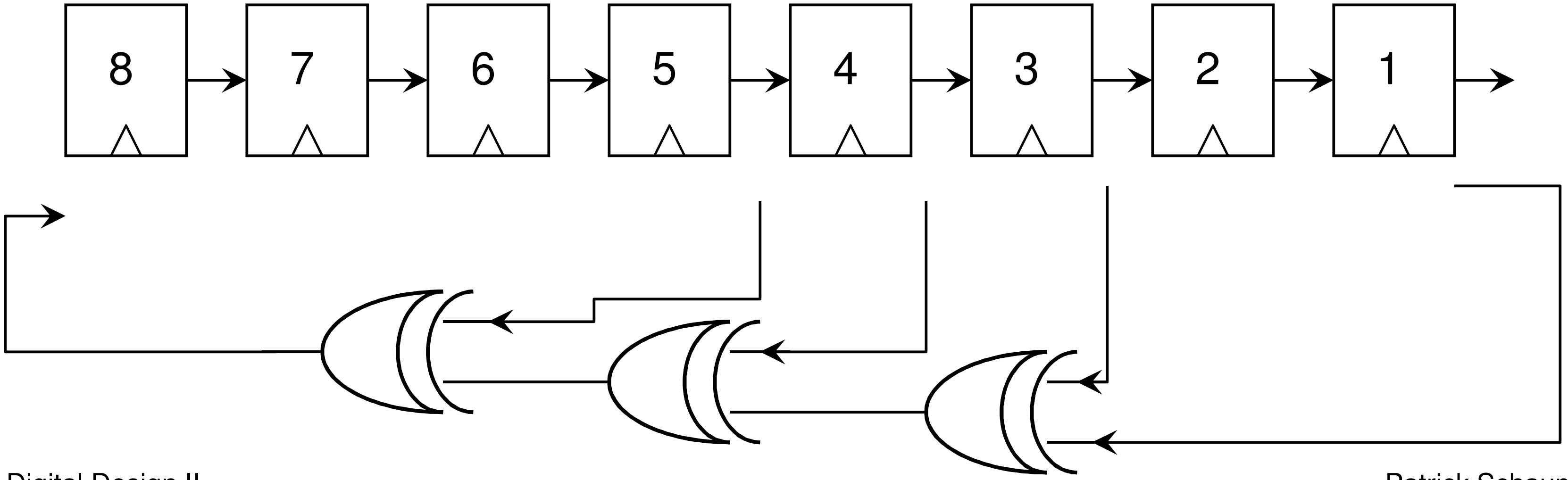
- Each Fibonacci LFSR can transform into Galois LFSR:
  - Reverse numbering of taps
  - Make XOR inputs XOR outputs and vice versa

- Example: starting with this Fibonacci LFSR



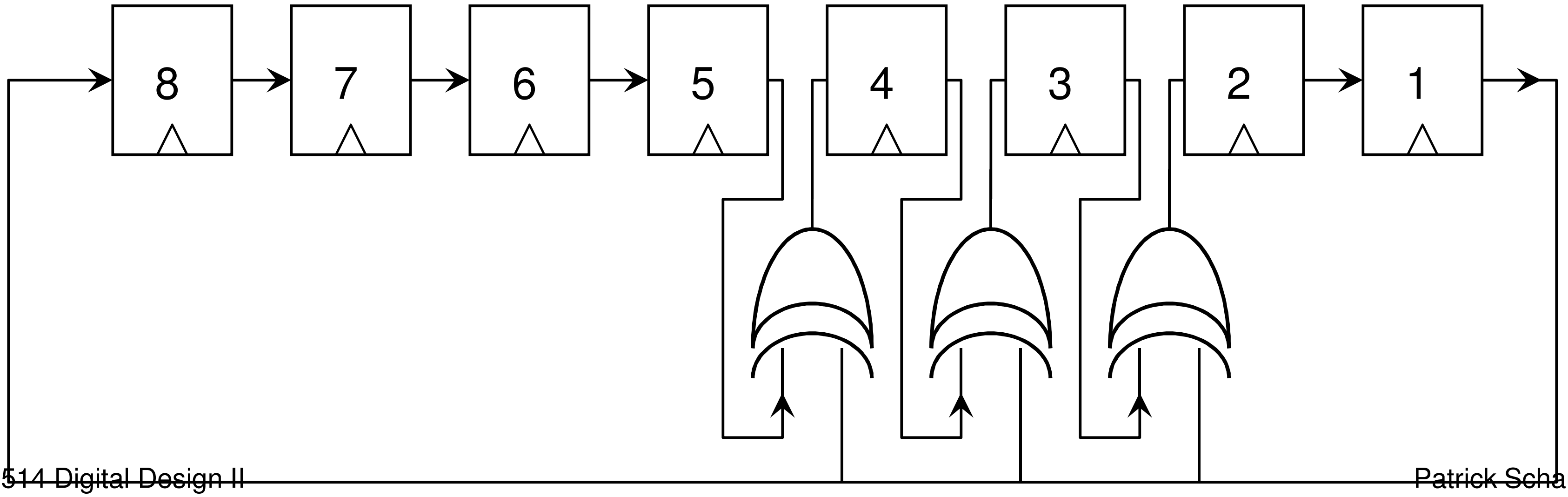
Fibonacci and Galois LFSR

- Disconnect XOR inputs
- Reverse tap numbering (not the direction of shifting!)



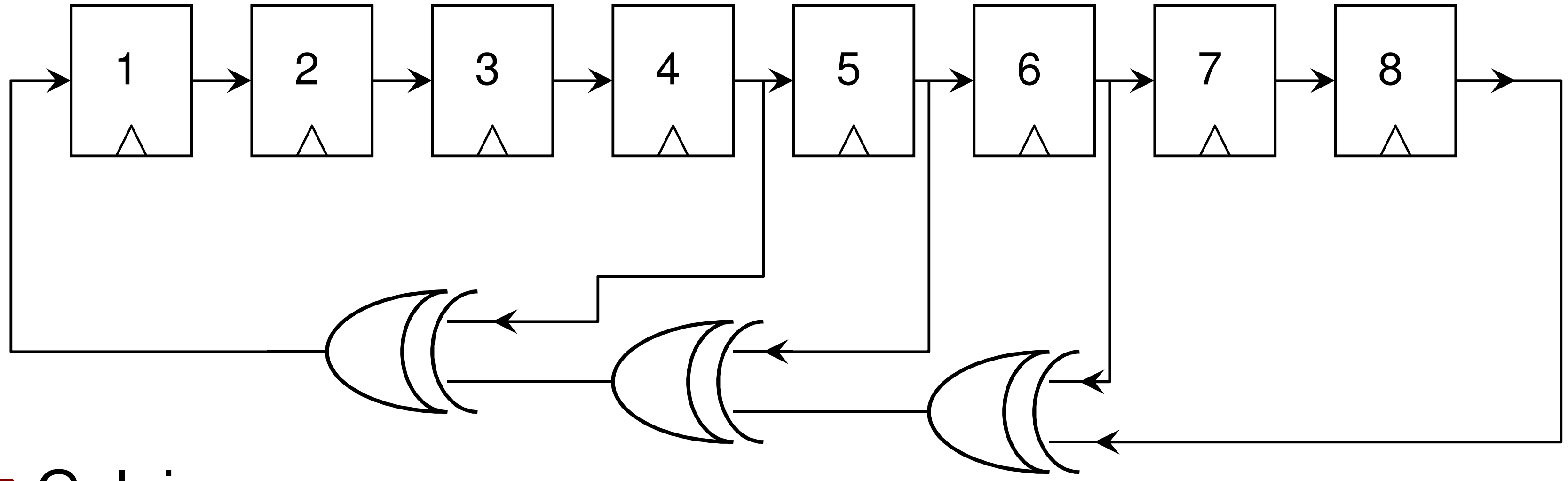
Fibonacci and Galois LFSR

- Turn XOR inputs into XOR outputs and vice versa

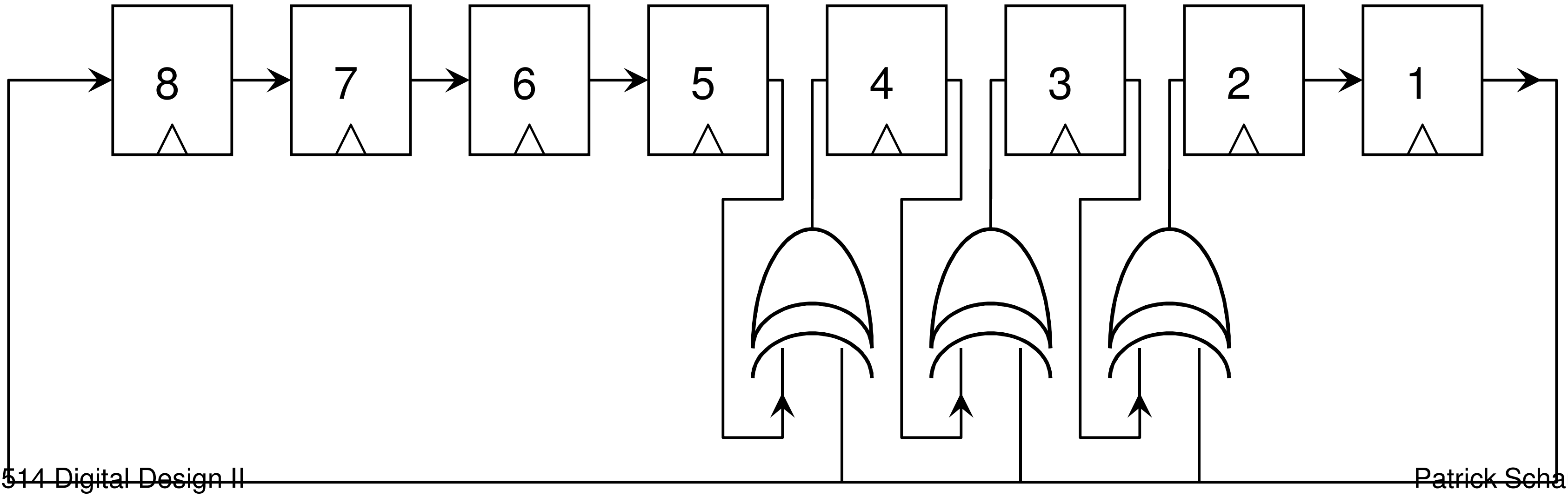


Which one is better for digital hardware?

- Fibonacci

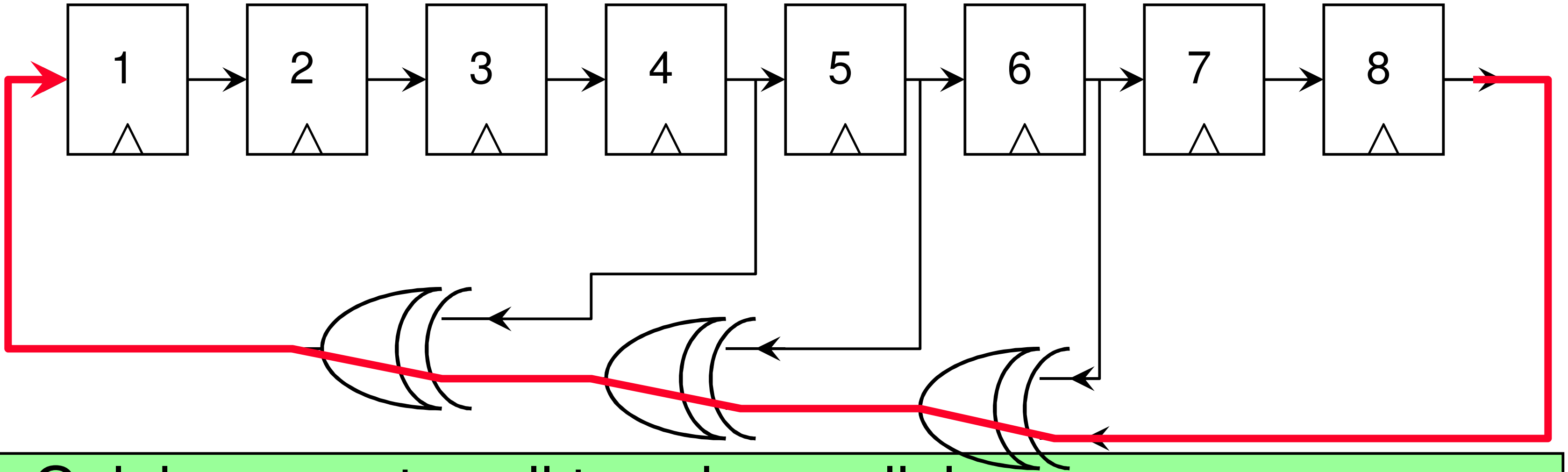


- Galois

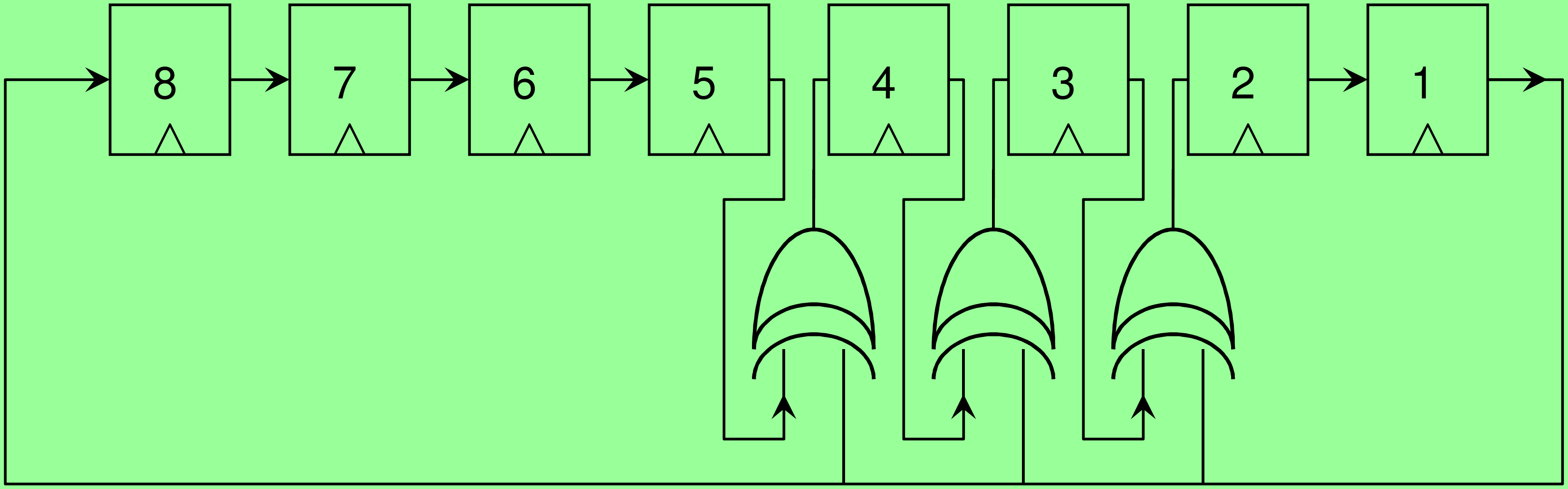


Which one is better for digital hardware?

- Fibonacci

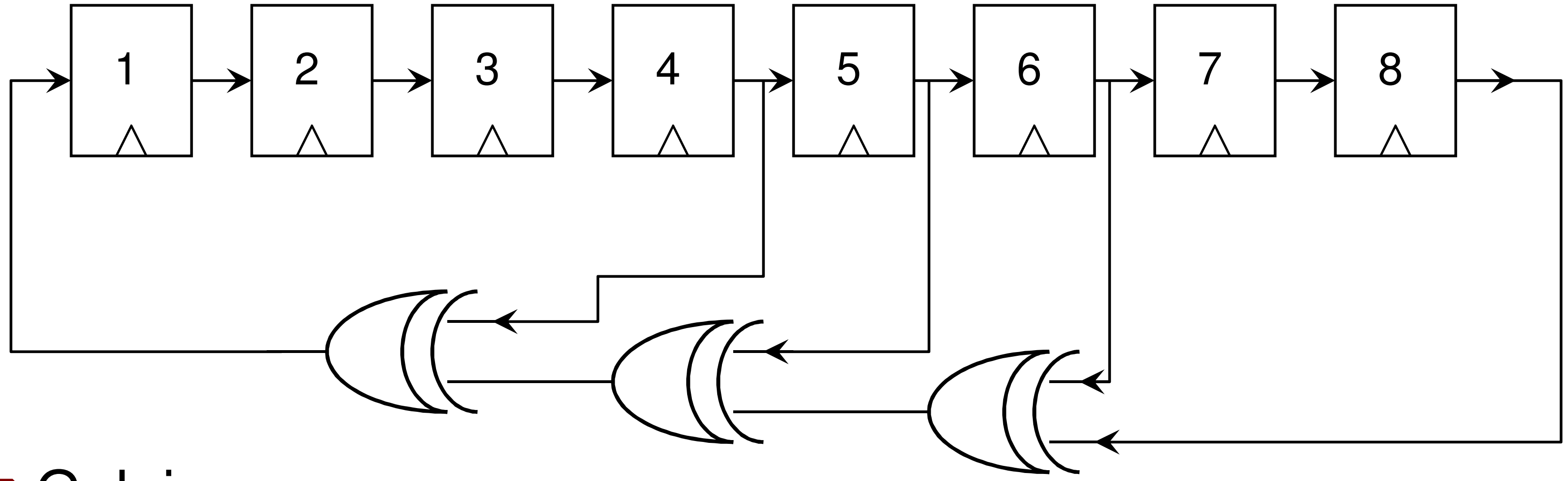


- Galois computes all taps in parallel

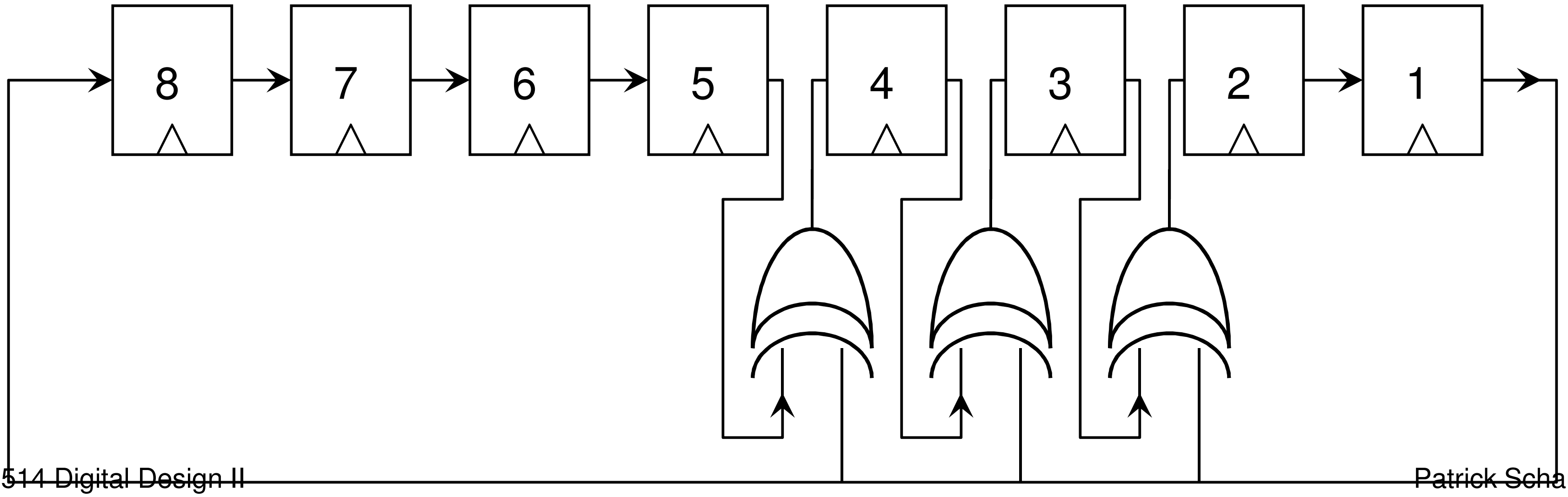


Which one is better for software?

- Fibonacci

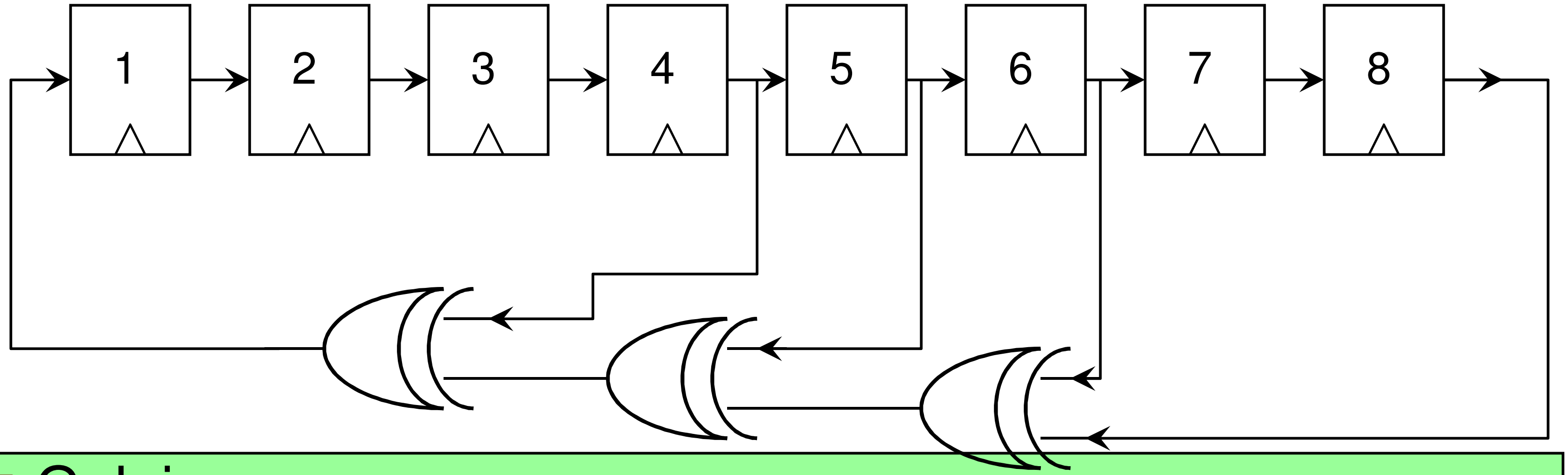


- Galois



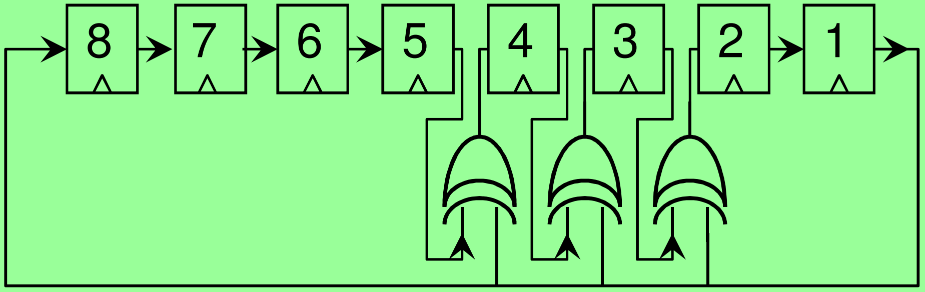
Which one is better for software?

- Fibonacci

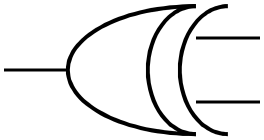


- Galois

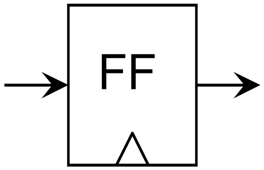
```
char_v = (char_v >> 1) ^ (-(signed char) (char_v & 1) & 0xe)
```



Let's write an LFSR in Verilog



xor(out, in1, in2)



You need to build this one
(structural, behavioral)

A Flip flop

```
module flipflop(q, clk, rst, d);
  input clk;
  input rst;
  input d;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
    begin
    if (rst)
      q = 0;
    else
      q = d;
    end
endmodule
```

A Flip flop

- Setup Time:
  - Time D has to be stable before a clock edge
- Hold Time:
  - Time D has to be stable after clock edge
- Propagation Delay:
  - Delay from clock edge to Q
  - Delay from reset to Q

How to specify propagation delay ?

A Flip flop

```
module flipflop(q, clk, rst, d);
  input clk;
  input rst;
  input d;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
    begin
    if (rst)
      q = 0;
    else
      q = d;
    end
endmodule
```

How to specify propagation delay ?

A Flip flop

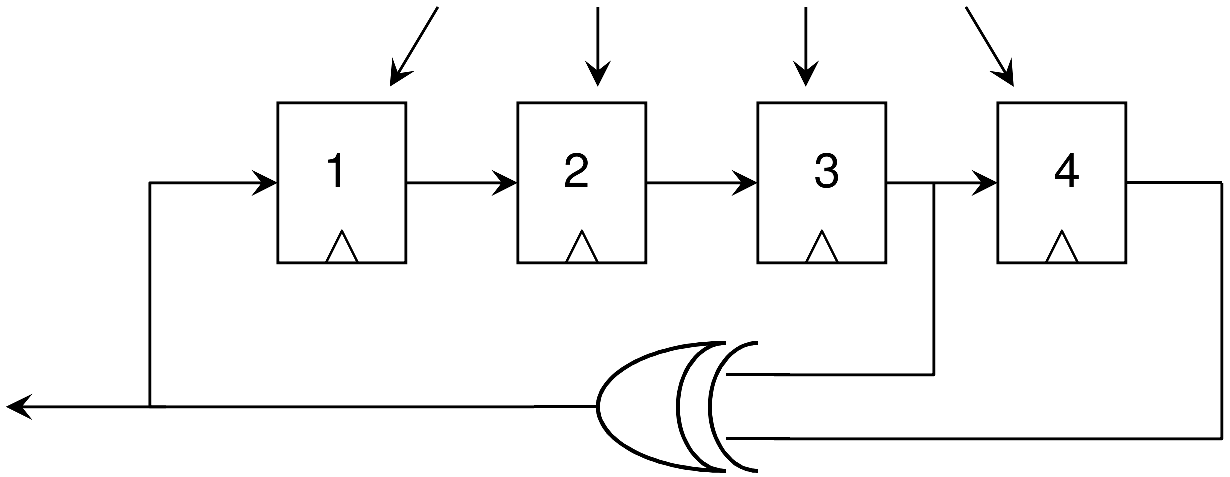
```
module flipflop(q, clk, rst, d);
  input clk;
  input rst;
  input d;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
    begin
    if (rst)
      #2 q = 0;
    else
      q = #3 d;
    end

  specify
    $setup(d, clk, 2);
    $hold(clk, d, 0);
  endspecify

endmodule
```

Test setup, hold

See Chapter 10 Palnitkar

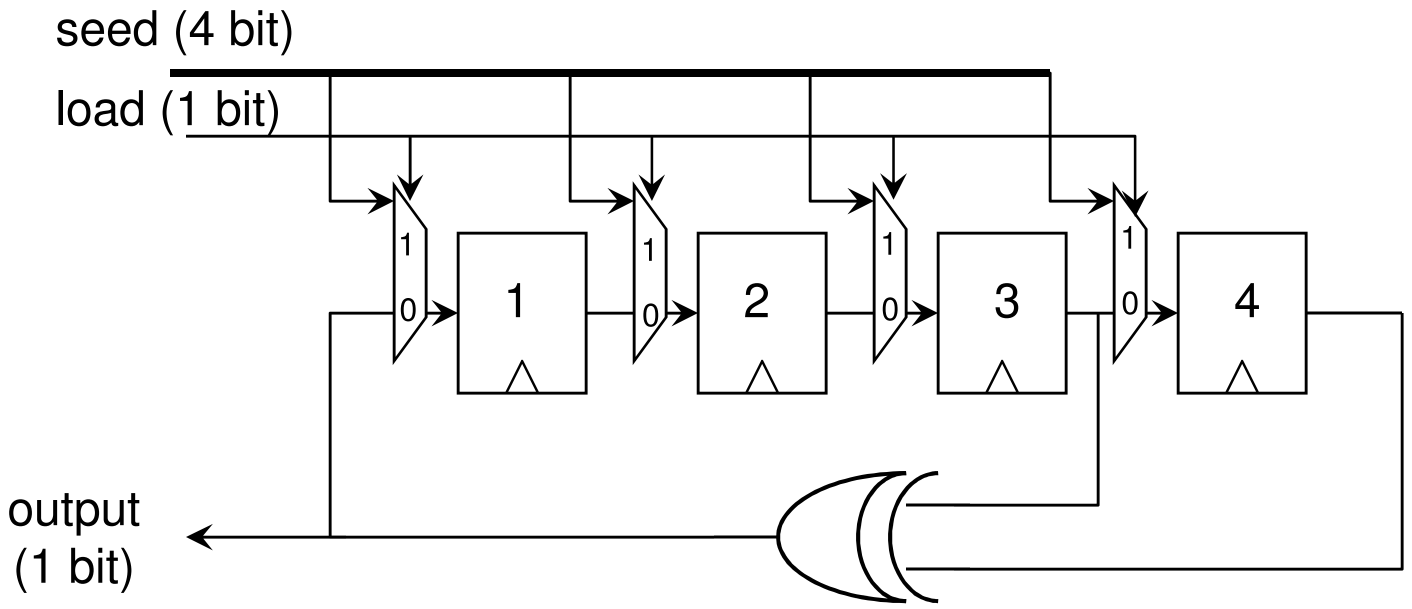
Let's turn the LFSR into a module

How to program this?

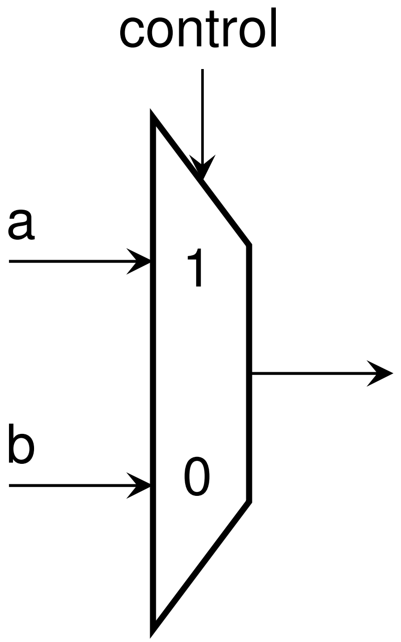


output
(1 bit)

Let's turn the LFSR into a module



Multiplexer symbol

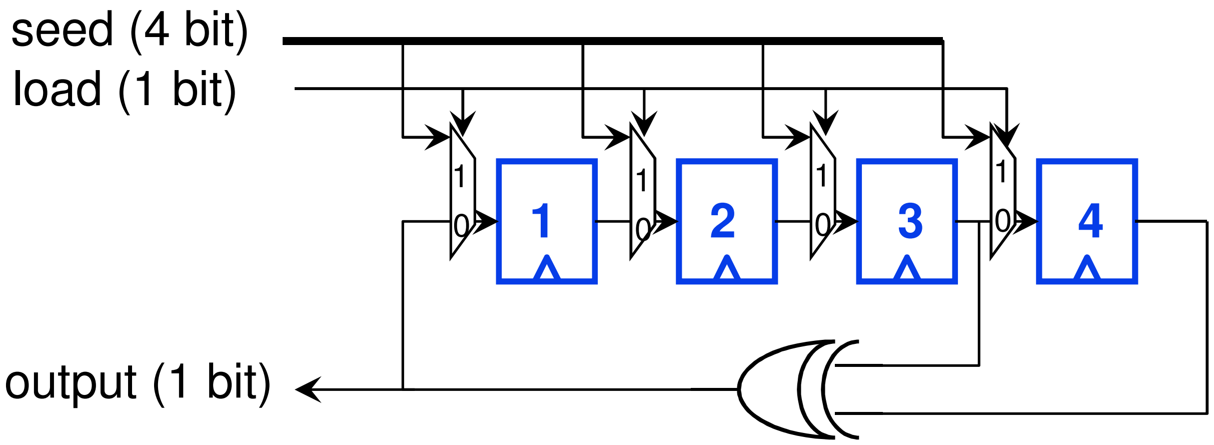


bit multiplexer

```
if (control)
  out = a;
else
  out = b;
```

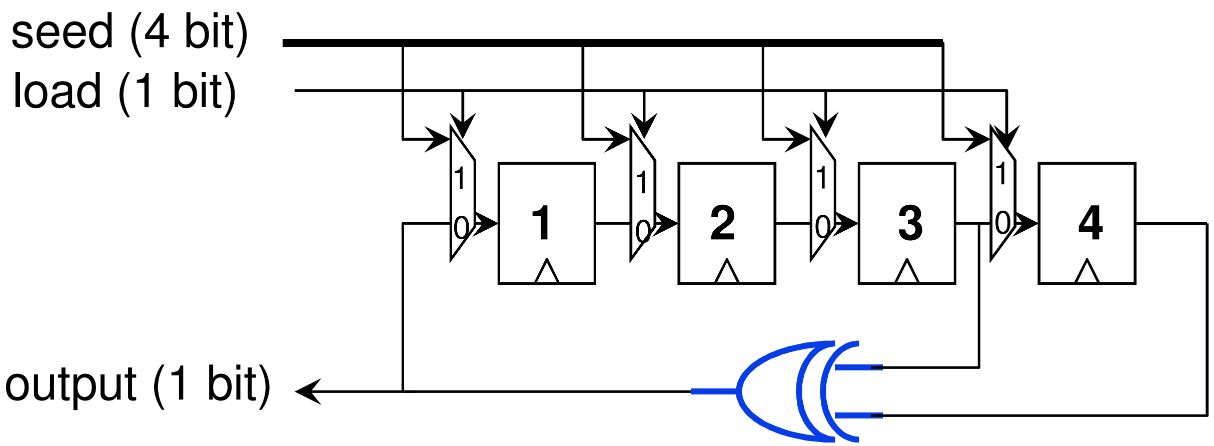
```
module mux(q, control, a, b);
  output q;
  reg q;
  input control, a, b;

  wire notcontrol;

  always @(control or
           notcontrol or
           a or b)
    q = (control & a) |
        (notcontrol & b);

  not (notcontrol, control);
endmodule;
```

LFSR, structural



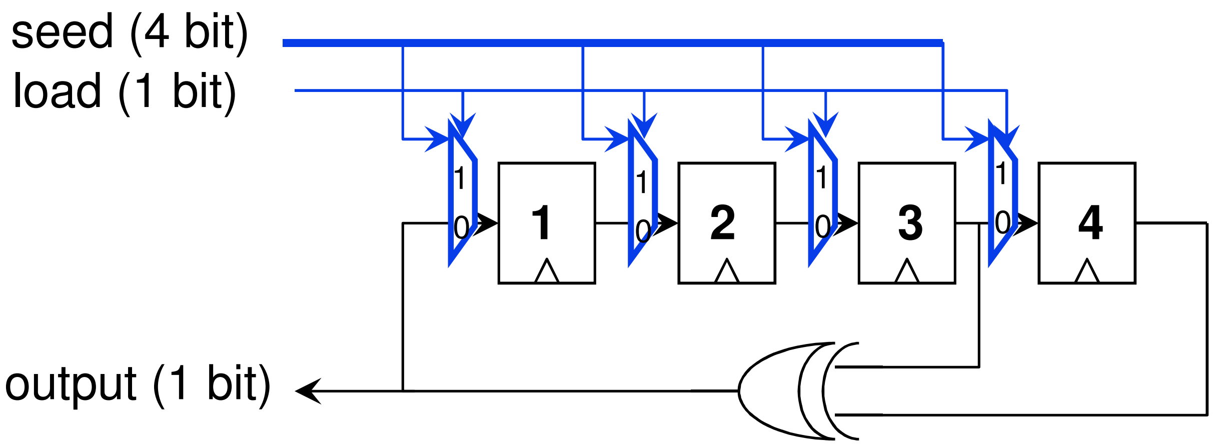
```
module lfsr(q, clk, rst, seed, load);
  ...
  wire [3:0] state_out;
  wire [3:0] state_in;

  flipflop F[3:0] (state_out, clk, rst, state_in);

endmodule
```

LFSR, structural



```
module lfsr(q, clk, rst, seed, load);
  ...
  wire [3:0] state_out;
  wire [3:0] state_in;
  wire nextbit;

  xor G1(nextbit, state_out[2], state_out[3]);

  assign q = nextbit;

endmodule
```

LFSR, structural



```
module lfsr(q, clk, rst, seed, load);
  ...
  wire [3:0] state_out;
  wire [3:0] state_in;
  wire nextbit;

  mux M1[3:0] (state_in, load, seed, {state_out[2],
                                      state_out[1],
                                      state_out[0],
                                      nextbit});

  assign q = nextbit;

endmodule
```

LFSR module - complete

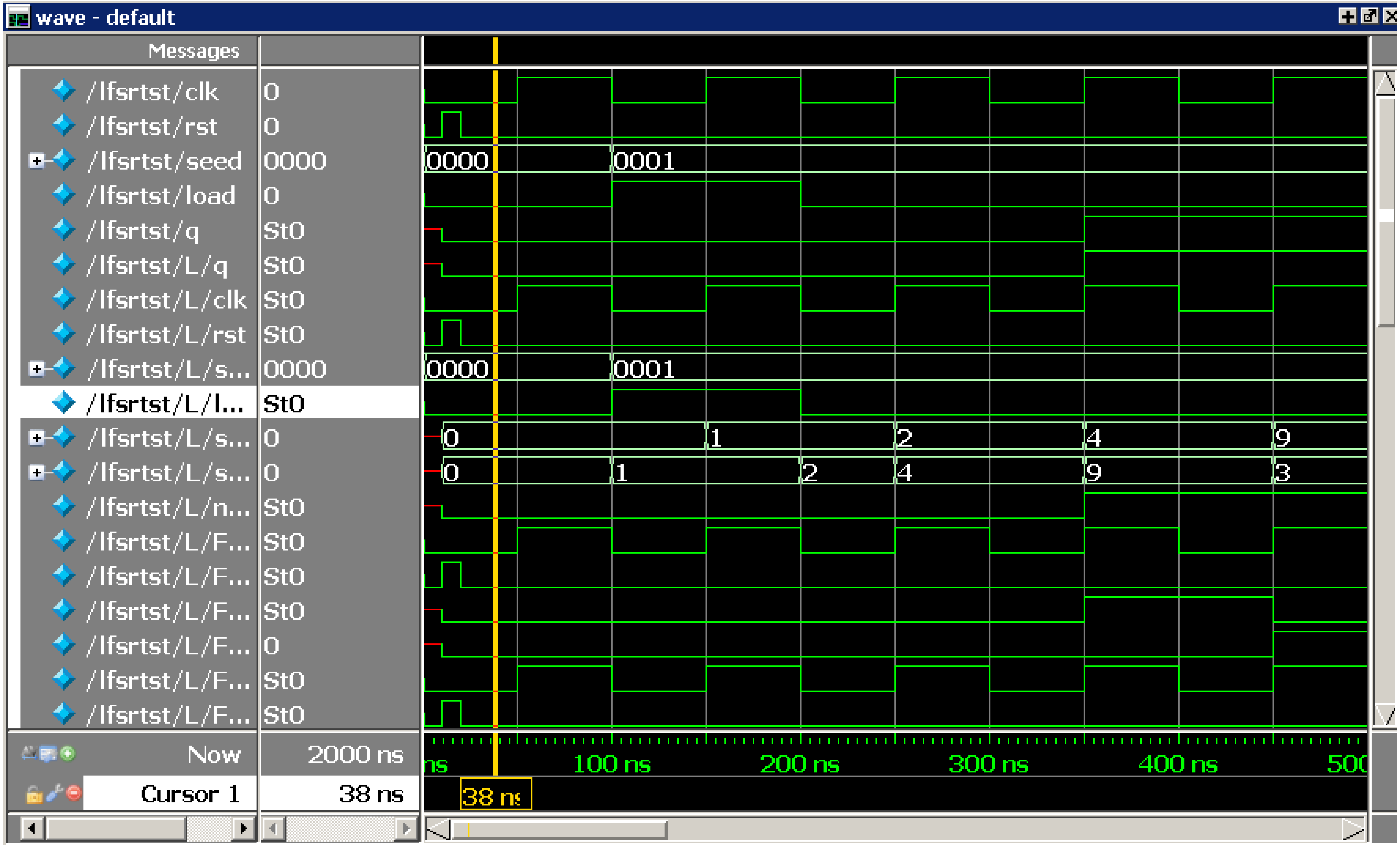
```
module lfsr(q, clk, rst, seed, load);
  output q;
  input [3:0] seed;
  input load;
  input rst;

  wire [3:0] state_out;
  wire [3:0] state_in;

  flipflop F[3:0] (state_out, clk, rst, state_in);
  mux M1[3:0] (state_in, load, seed, {state_out[2],
                                      state_out[1],
                                      state_out[0],
                                      nextbit});
  xor G1(nextbit, state_out[2], state_out[3]);
  assign q = nextbit;

endmodule
```

LFSR testbench

```
module lfsrtst;
    reg clk;
    reg rst;
    reg [3:0] seed;
    reg load;
    wire q;
    lfsr L(q, clk, rst,
           seed, load);

    // initialization
    // apply reset pulse
    initial
      begin
      clk = 0;
      load = 0;
      seed = 0;
      rst = 0;
      #10 rst = 1;
      #10 rst = 0;
    end
```

```
    // drive clock
    always
    #50 clk = !clk;

    // program lfsr
    initial begin
    #100 seed = 4'b0001;
         load = 1;
    #100 load = 0;
    end

endmodule
```

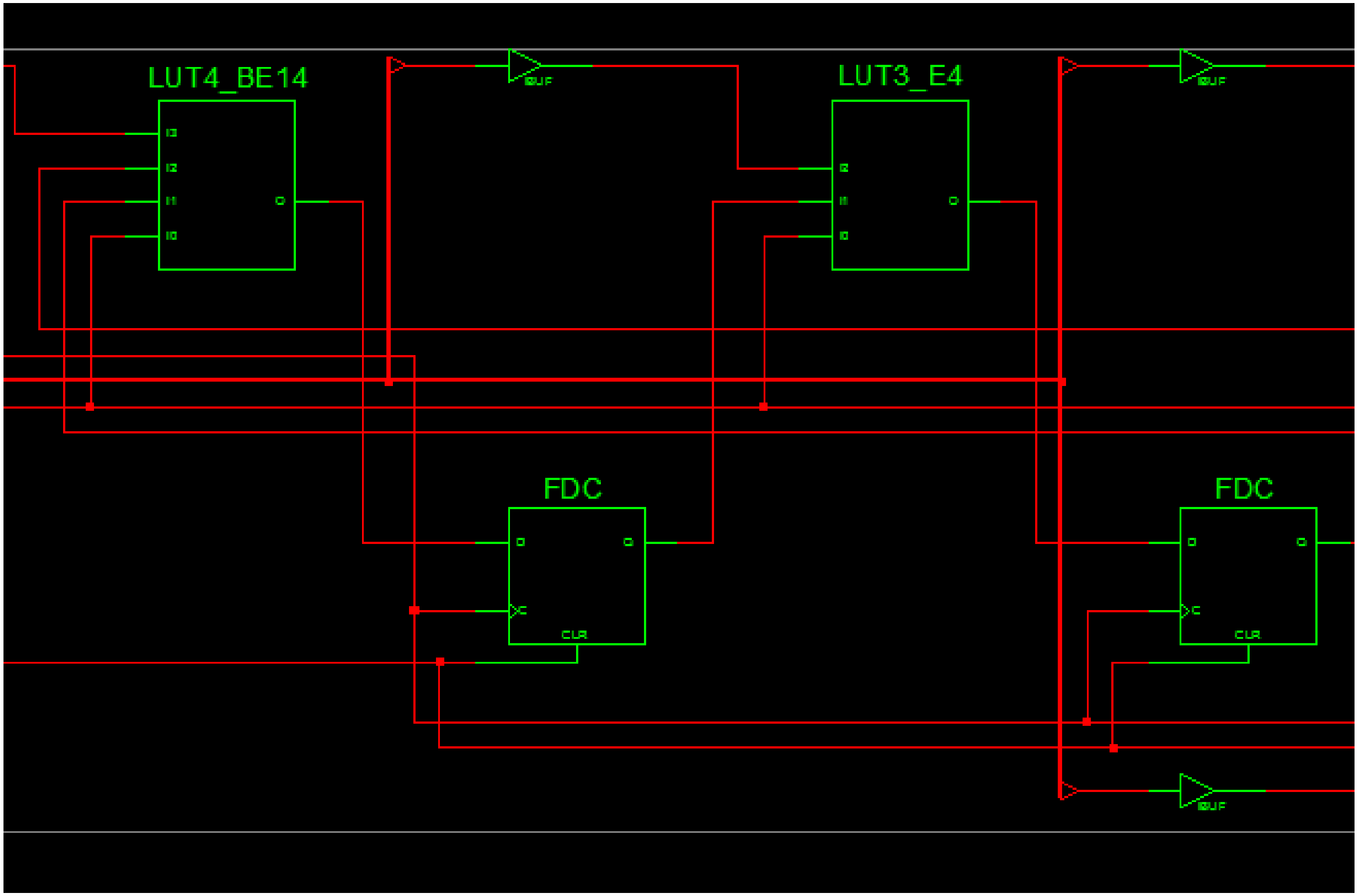
Simulation ..



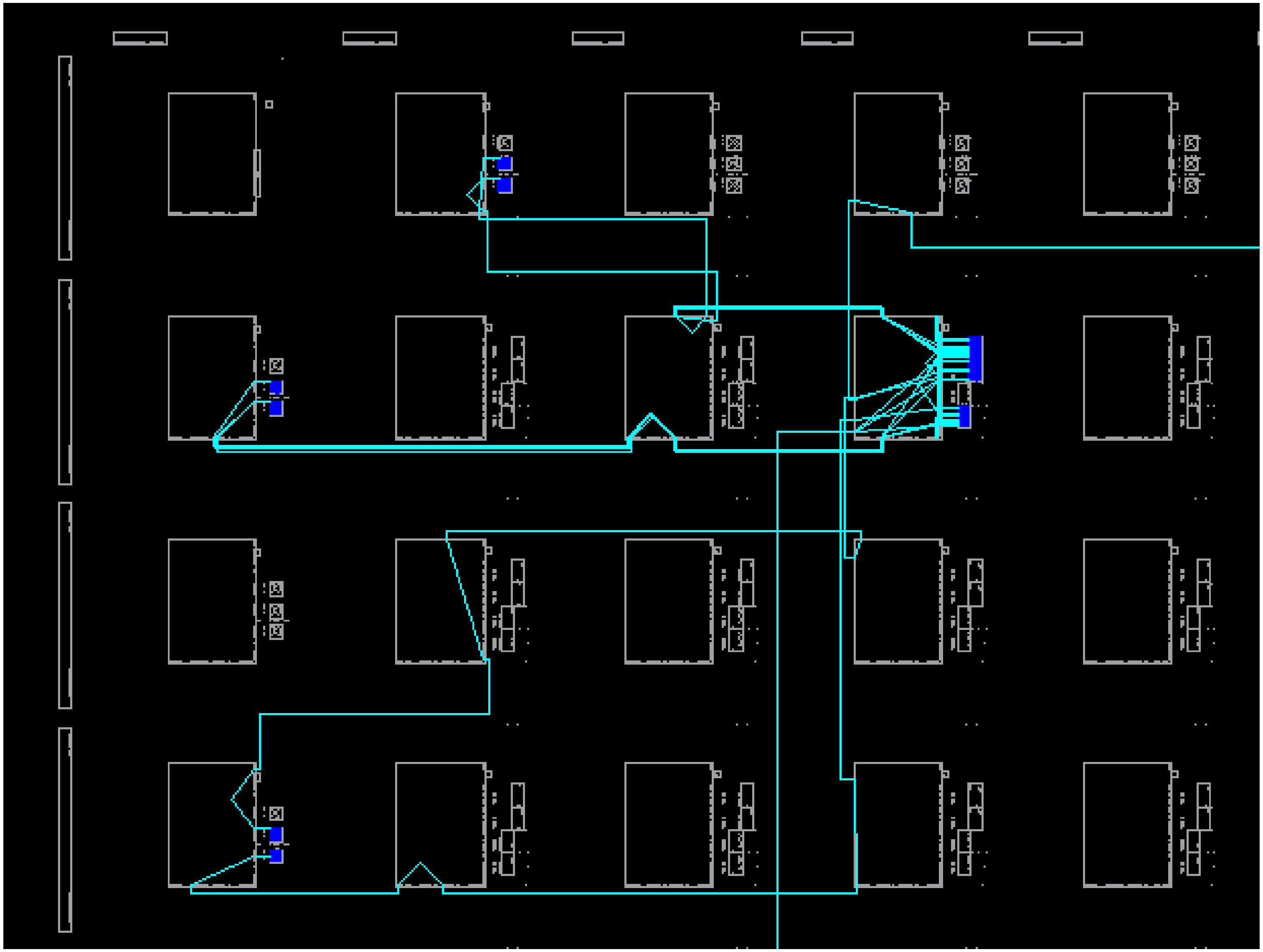
wave - default

Messages	
/lfsrtst/clk	0
/lfsrtst/rst	0
/lfsrtst/seed	0000
/lfsrtst/load	0
/lfsrtst/q	St0
/lfsrtst/L/q	St0
/lfsrtst/L/clk	St0
/lfsrtst/L/rst	St0
/lfsrtst/L/s...	0000
/lfsrtst/L/l...	St0
/lfsrtst/L/s...	0
/lfsrtst/L/s...	0
/lfsrtst/L/n...	St0
/lfsrtst/L/F...	St0
/lfsrtst/L/F...	St0
/lfsrtst/L/F...	St0
/lfsrtst/L/F...	0
/lfsrtst/L/F...	St0
/lfsrtst/L/F...	St0
Now	2000 ns
Cursor 1	38 ns

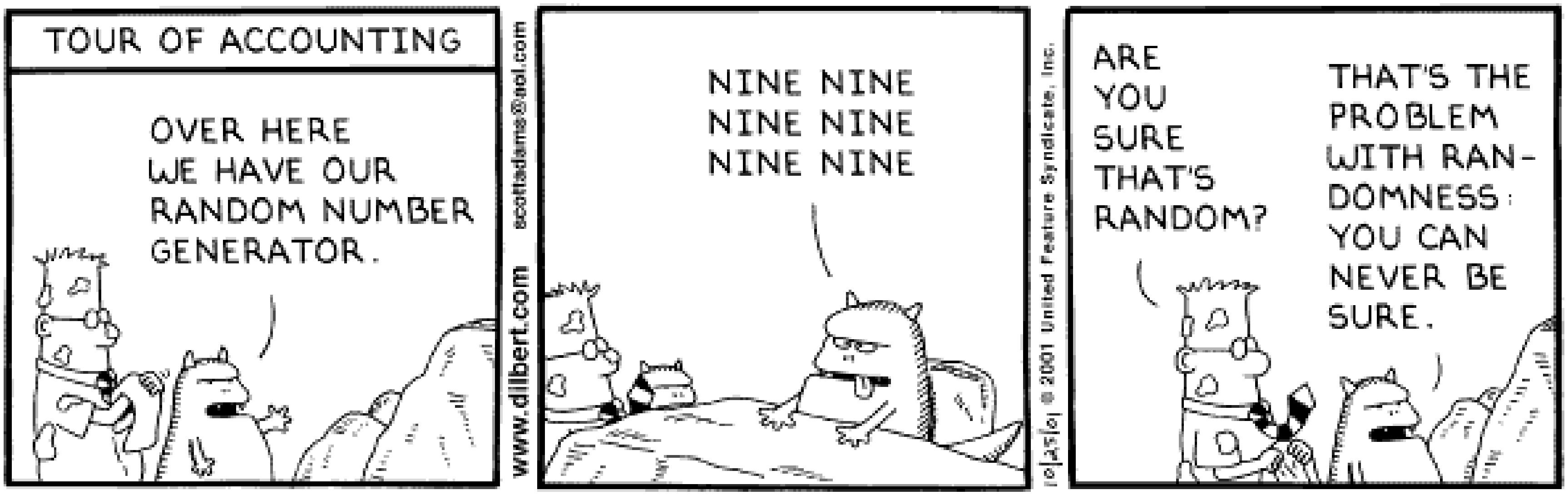
Synthesis ..


LUT4_BE14
I3
I2
I1
I0
O
BUF
LUT3_E4
I2
I1
I0
O
BUF
FDC
D
Q
C
CLR
FDC
D
Q
C
CLR
BUF

Place and Route ..

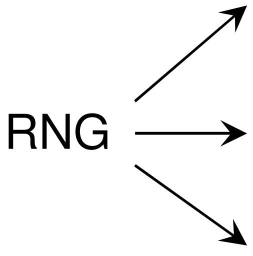


Can a random number generator have flaws?


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
scottadams@aol.com
www.dilbert.com
NINE NINE NINE NINE NINE NINE
10/25/01 © 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

Can a random number generator have flaws?

- Problem #1: it can have bias
  - Meaning: a certain number occurs more often then others
  - Expressed in the *entropy rate* of the generator
  - Entropy = true information rate (in bit/sec), can be lower then the actual bitrate of the random number generator
  - Example: Assume an RNG that produces three events A,B,C encoded with two bits



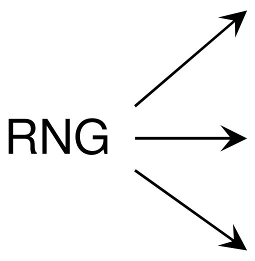
symbol	probability	bitpattern
A	P(A) = 1/2	00
B	P(B) = 1/4	01
C	P(C) = 1/4	10

E.g. ABACAABC... is encoded as 0001001000000110...

So this bitstream has much more '0' then '1'. It has a bias.

Can a random number generator have flaws?

- Problem #1: it can have bias
  - Meaning: a certain number occurs more often then others
  - Expressed in the *entropy rate* of the generator
  - Entropy = true information rate (in bit/sec), can be lower then the actual bitrate of the random number generator
  - Example: Assume an RNG that produces three events A,B,C encoded with two bits



symbol	probability	**better bitpattern**
A	P(A) = 1/2	**0**
B	P(B) = 1/4	**10**
C	P(C) = 1/4	**11**

E.g. ABACAABC... is encoded as 010011001011...

In this bitstream, the number of '1' and '0' are <u>balanced</u>.

Can a random number generator have flaws?

- Problem #1: it can have bias
    - Do LFSR have a bias ?

Can a random number generator have flaws?

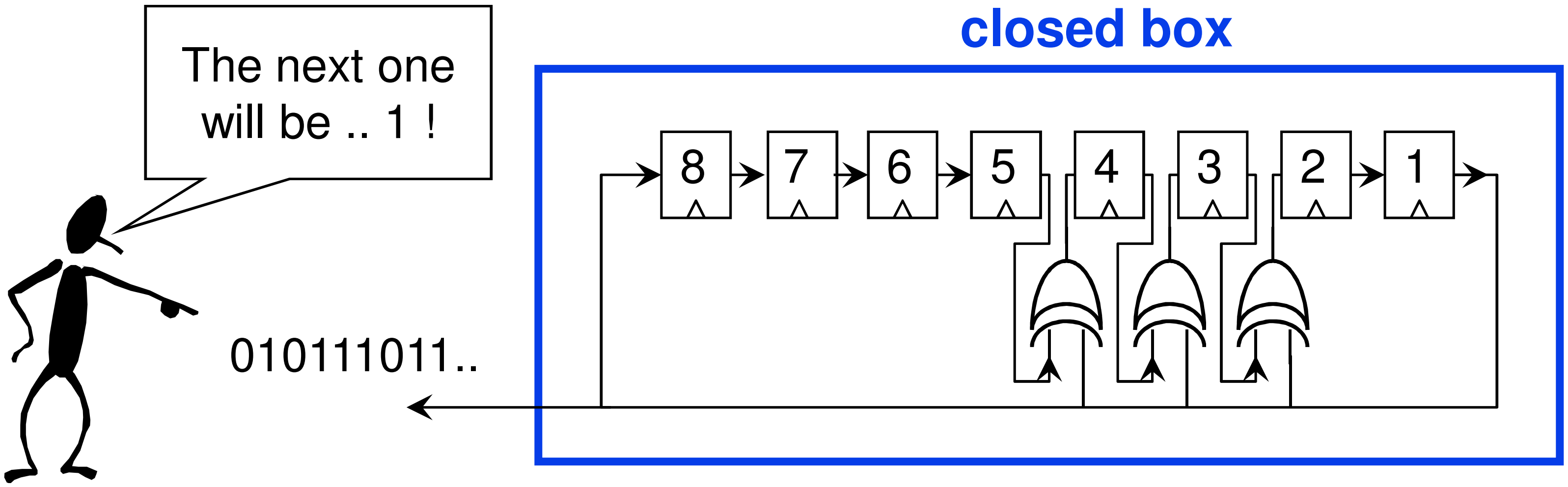
- Problem #1: it can have bias
  - Do LFSR have a bias ?
  - Yes, they have a small bias because the all-zero state never appears.
  - However, for a very long LFSR, the bias becomes negligible

Can a random number generator have flaws?

- Problem #2: it can be predicted
  - Not when truly random phsysical phenomena
  - But, if it is a Pseudo RNG (like an LFSR), it is a deterministic sequence.
  - Is this really a problem? Yes!
    - Don't want to use a predictable RNG for dealing cards, driving a slot machine, ... (at least not if you own the place).
    - Don't want to use predicatable RNG in security. Predicability = weakness

Can a random number generator have flaws?

- Problem #2: it can be predicted
  - The real issue for PRNG is: can the value of bit N+1 be predicted when someone observes the first N bits.



LFSR are *very* predictable ..

- Predicting the next output bit is equivalent to knowing the feedback pattern of the LFSR and the states of all LFSR flip-flops.

- Mathematicians (Berlekamp-Massey) found that:
  - Given an N-bit LFSR with unknown feedback pattern, then only 2N bits are needed to predict bit 2N + 1

- So let's say we have and 8-bit LFSR, then we need only 16 bits of the RNG stream before it becomes predicatble

- **LFSR are *unsuited* for everything that should be unpredictable**

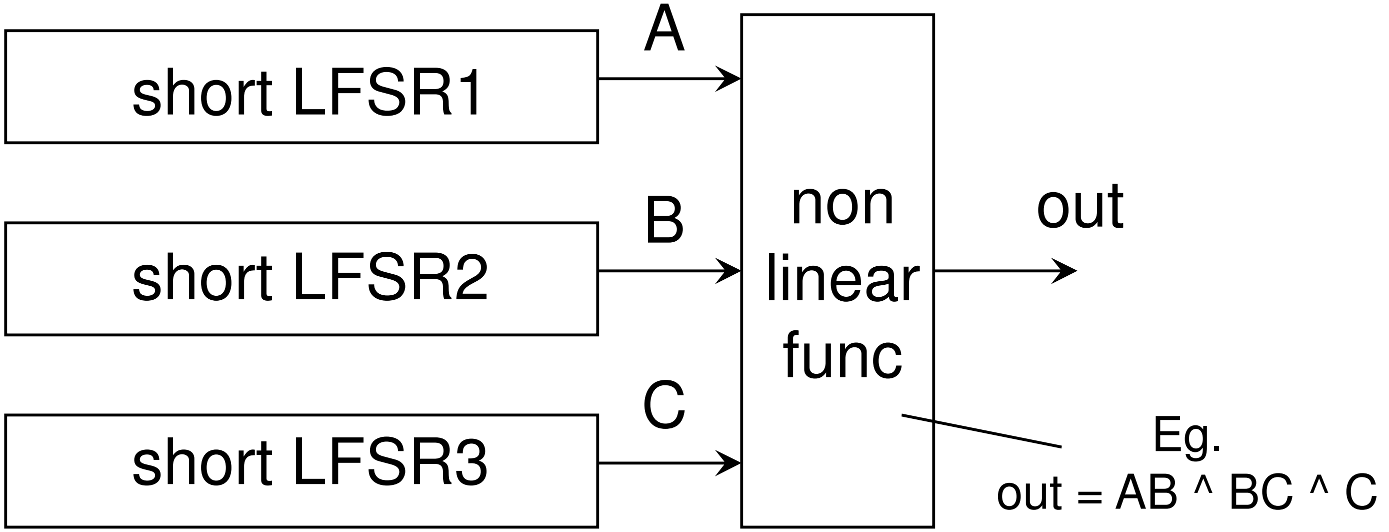
To be unpredictable, the LFSR should be long

- Solution 1

Use a maximal-length P(x) = x^N + ... + 1
with N >>> (E.g. 65,536)

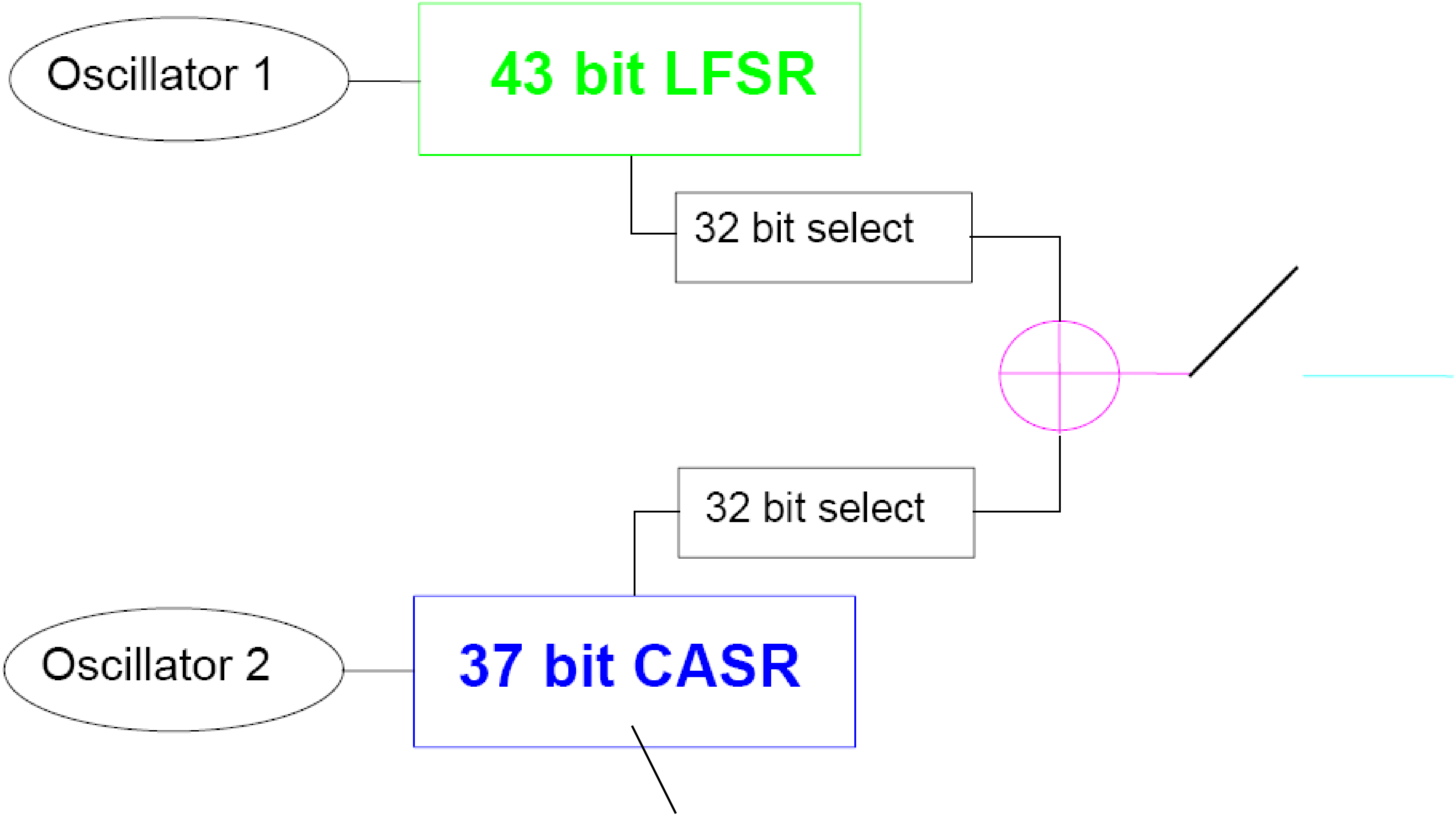
Very expensive to make! 64K flip-flops ..

- Solution 2: Non-linear Combination Generator



Eg.
out = AB ^ BC ^ C

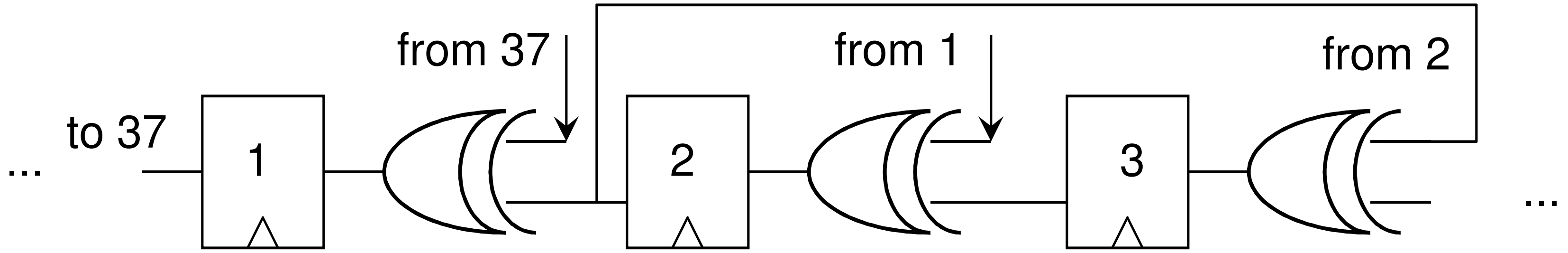
Example design by Tkacik, 2002



Cellular Automata Shift Register

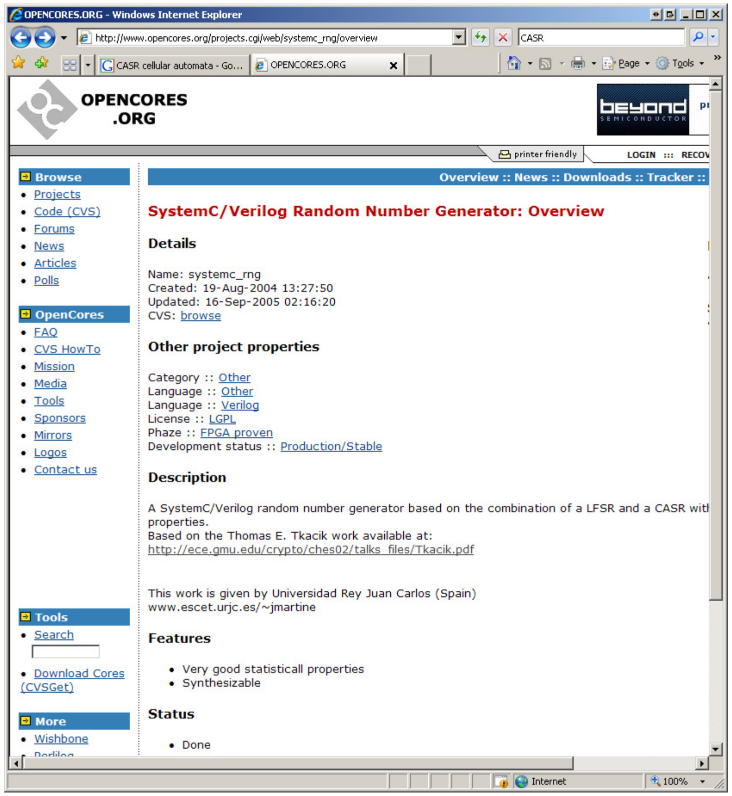
Example design by Tkacik, 2002

- 43-bit LFSR defined by
  - P(X) = X^43 + X^41 + X^20 + X + 1 => 3XOR, 43 taps
  - Maximal Length: 2^43-1
  - Bias ~ 2^-43 (because all-zero pattern cannot appear)
- 37-bit cellular automata shift register
  - Combines previous and next statereg into current state reg
  - Similar to 37 intertwined state machines (automata)
  - Maximal Length: 2^37-1
  - Bias ~ 2^-37



Sample Implementation

- On opencores you can find an implementation of Tkacik's design - *assigned reading of today*
  - (this design has a few minor differences with the spec written by Tkacik - but OK for our purpose)



SystemC/Verilog Random Number Generator: Overview

Details

Name: systemc_rng
Created: 19-Aug-2004 13:27:50
Updated: 16-Sep-2005 02:16:20
CVS: browse

Other project properties

Category :: Other
Language :: Other
Language :: Verilog
License :: LGPL
Phaze :: FPGA proven
Development status :: Production/Stable

Description

A SystemC/Verilog random number generator based on the combination of a LFSR and a CASR wit
properties.
Based on the Thomas E. Tkacik work available at:
http://ece.gmu.edu/crypto/ches02/talks_files/Tkacik.pdf

This work is given by Universidad Rey Juan Carlos (Spain)
www.escet.urjc.es/~jmartine

Features

- Very good statisticall properties
- Synthesizable

Status

- Done

Module interface

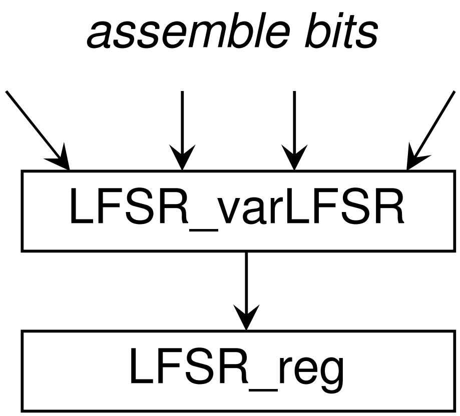
```
module rng(clk,reset,loadseed_i,seed_i,number_o);
input clk;
input reset;
input loadseed_i;
input [31:0] seed_i;
output [31:0] number_o;
reg [31:0] number_o;

reg [42:0] LFSR_reg;  // internal state
reg [36:0] CASR_reg;  // internal state


always (.. CASR ..)

always (.. LFSR ..)

always (.. combine outputs ..)


endmodule
```

LFSR Part

```
reg[42:0] LFSR_varLFSR; // temporary working var
reg outbitLFSR;         // temporary working var
always @(posedge clk or negedge reset)
   begin
   if (!reset )
      begin
      ...
      end
   else
      begin
      if (loadseed_i )
         begin
         ...
         end
      else
         begin
         ...
         end
      end
   end
```

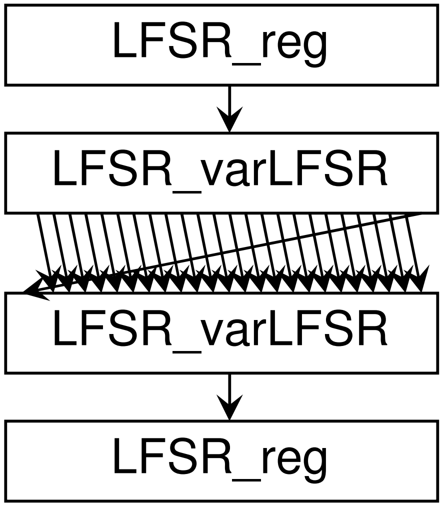
LFSR Part

```
reg[42:0] LFSR_varLFSR; // temporary working var
reg outbitLFSR;         // temporary working var
always @(posedge clk or negedge reset)
   begin
   if (!reset )
      begin
      LFSR_reg  = (1);
      end
   else
       begin
        if (loadseed_i )
         begin
         LFSR_varLFSR [42:32]=0;
         LFSR_varLFSR [31:0]=seed_i ;
         LFSR_reg  = (LFSR_varLFSR );
         end
       else
         begin
         ...
         end
      end
   end
```



LFSR Part

```
reg[42:0] LFSR_varLFSR; // temporary working var
reg outbitLFSR;         // temporary working var
always @(posedge clk or negedge reset)
   begin
   if (!reset )
   else
       begin
        if (loadseed_i )
        else
         begin
         LFSR_varLFSR = LFSR_reg;
         LFSR_varLFSR [42] = LFSR_varLFSR [41];
         outbitLFSR        = LFSR_varLFSR [42];
         LFSR_varLFSR [42] = LFSR_varLFSR [41];
         LFSR_varLFSR [41] = LFSR_varLFSR [40]^outbitLFSR ;
         // some lines skipped ...
         LFSR_varLFSR [0]  = LFSR_varLFSR [42];
         LFSR_reg     = LFSR_varLFSR;
         end
      end
   end
```



CASR Part (similar ...)

```
//CASR:
reg[36:0] CASR_varCASR,CASR_outCASR;
always @(posedge clk or negedge reset)
   begin
   if (!reset )
      begin
      ...
      end
   else
      begin
      if (loadseed_i )
         begin
         ...
         end
      else
         begin
         ...
         end
      end
   end
```

CASR Part (similar ...)

```
//CASR:
reg[36:0] CASR_varCASR,CASR_outCASR; // temp
always @(posedge clk or negedge reset)
   begin
   if (!reset )
      begin
      CASR_reg = 1;
      end
   else
      begin
      if (loadseed_i )
         begin
         CASR_varCASR [36:32]= 0;
         CASR_varCASR [31:0] = seed_i ;
         CASR_reg  = (CASR_varCASR );
         end
      else
         begin
         ...
         end
      end
   end
```

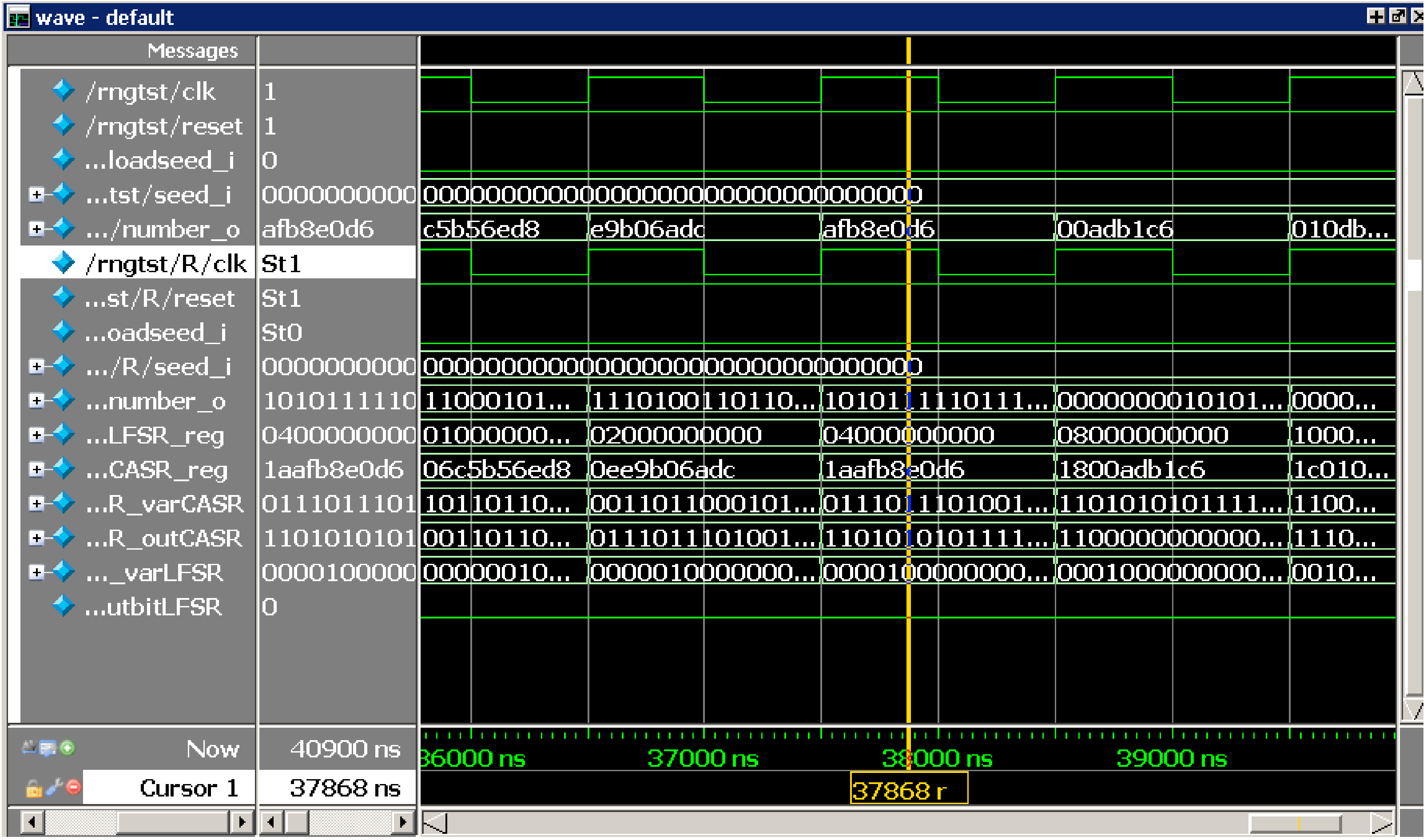
CASR Part (similar ...)

```
//CASR:
reg[36:0] CASR_varCASR,CASR_outCASR; // temp
always @(posedge clk or negedge reset)
   begin
   if (!reset )
   else
      begin
      if (loadseed_i )
      else
         begin
         CASR_varCASR     = CASR_reg ;
         CASR_outCASR [36]= CASR_varCASR [35]^CASR_varCASR [0];
         CASR_outCASR [35]= CASR_varCASR [34]^CASR_varCASR [36];
         // ... some lines skipped
         CASR_reg         = CASR_outCASR;
         end
      end
   end
```

Combine outputs

```
always @(posedge clk or negedge reset)
   begin
   if (!reset )
      begin
      number_o  = (0);
      end
   else
      begin
      number_o  = (LFSR_reg [31:0]^CASR_reg[31:0]);
      end
   end
```

Simulation ..

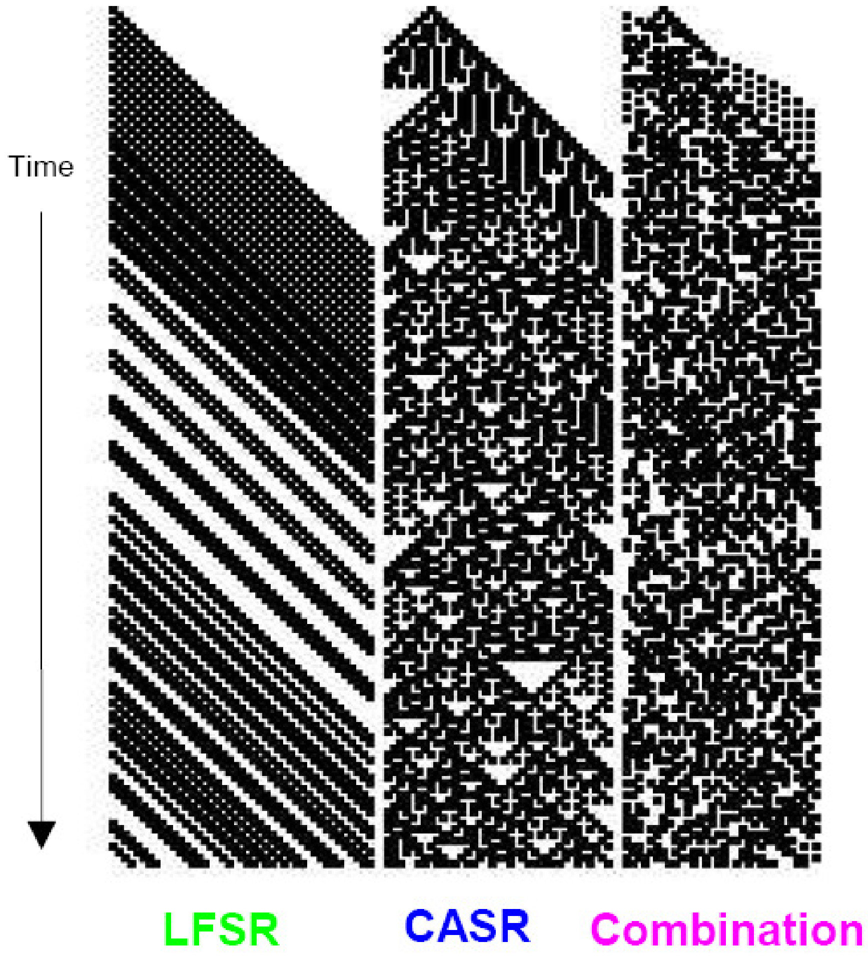


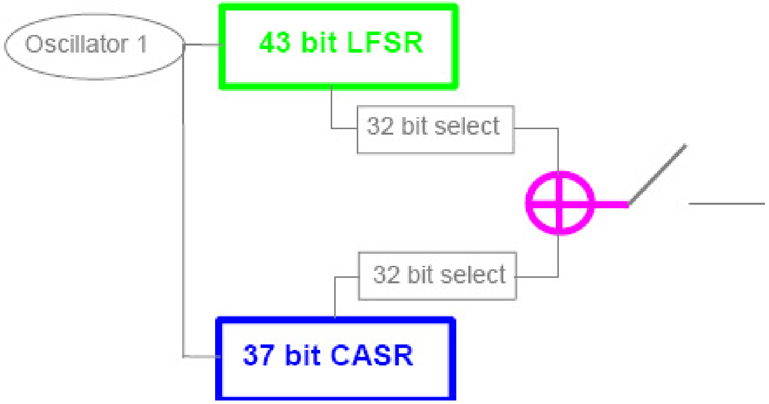
wave - default

Messages						
/rngtst/clk	1					
/rngtst/reset	1					
...loadseed_i	0					
...tst/seed_i	0000000000	00000000000000000000000000000000				
.../number_o	afb8e0d6	c5b56ed8	e9b06adc	afb8e0d6	00adb1c6	010db...
/rngtst/R/clk	St1					
...st/R/reset	St1					
...oadseed_i	St0					
.../R/seed_i	0000000000	00000000000000000000000000000000				
...number_o	1010111110	11000101...	1110100110110...	1010111110111...	0000000010101...	0000...
...LFSR_reg	0400000000	01000000...	02000000000	04000000000	08000000000	1000...
...CASR_reg	1aafb8e0d6	06c5b56ed8	0ee9b06adc	1aafb8e0d6	1800adb1c6	1c010...
...R_varCASR	0111011101	10110110...	0011011000101...	0111011101001...	1101010101111...	1100...
...R_outCASR	1101010101	00110110...	0111011101001...	1101010101111...	1100000000000...	1110...
..._varLFSR	0000100000	00000010...	0000010000000...	0000100000000...	0001000000000...	0010...
...utbitLFSR	0					

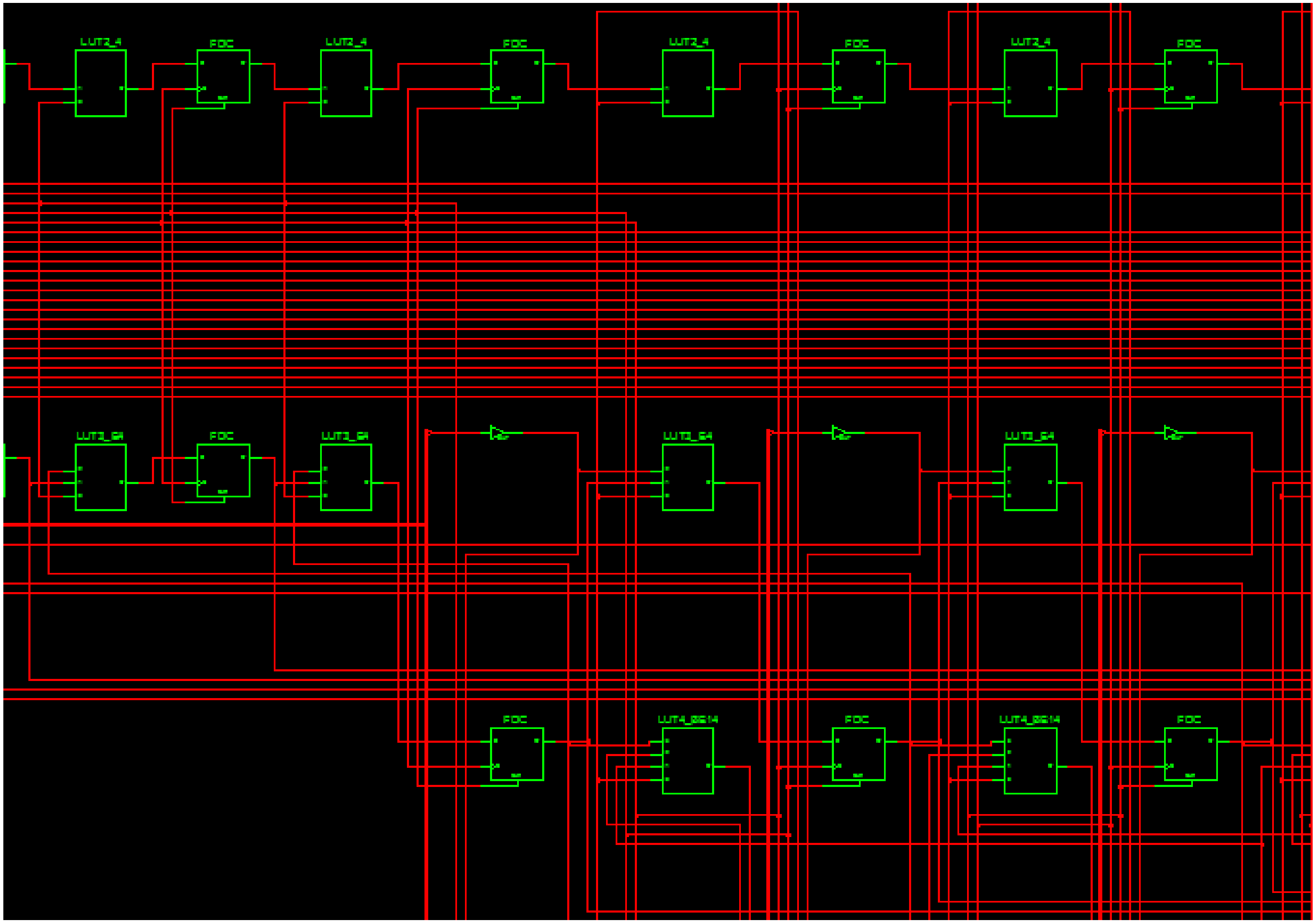
Now	40900 ns	
Cursor 1	37868 ns	37868 r

Simulation (looking at the state registers)

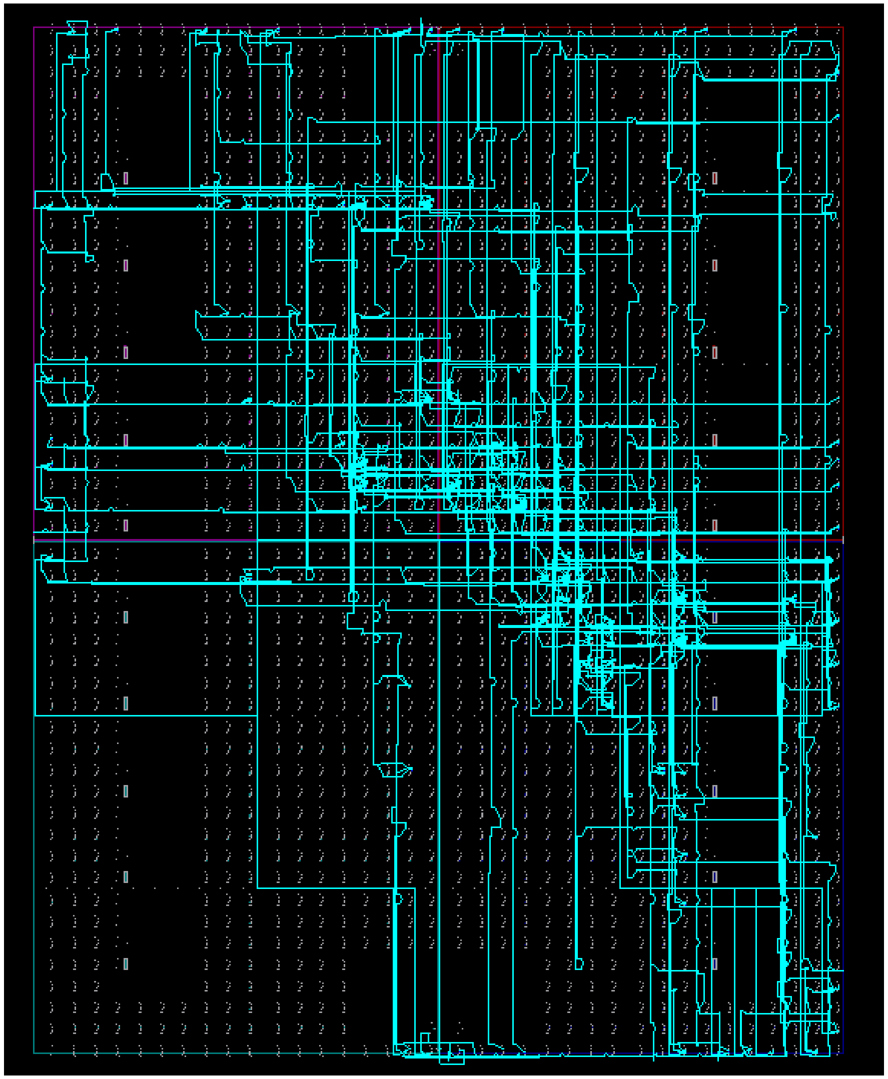




Synthesis ..


LUT2_4
FDC
LUT2_4
FDC
LUT2_4
FDC
LUT2_4
FDC
LUT3_EA
FDC
LUT3_EA
LUT3_E4
LUT3_E4
FDC
LUT4_8614
FDC
LUT4_8614
FDC

Place and Route ..



Summary

- ❑ Random number generators
  - Many useful applications
- ❑ Linear Feedback Shift Registers: PRNG
  - Fibonacci and Galois
  - Maximal-length LFSR
  - Structural Verilog Model
- ❑ Flaws of Random Number Generators
  - Bias
  - Predictability
- ❑ Nonlinear Combination Generator
  - Design by Tkacik
  - Behavioral Verilog Model