Cryptology
Lecture 2

Stream Ciphers

Joseph Phillips

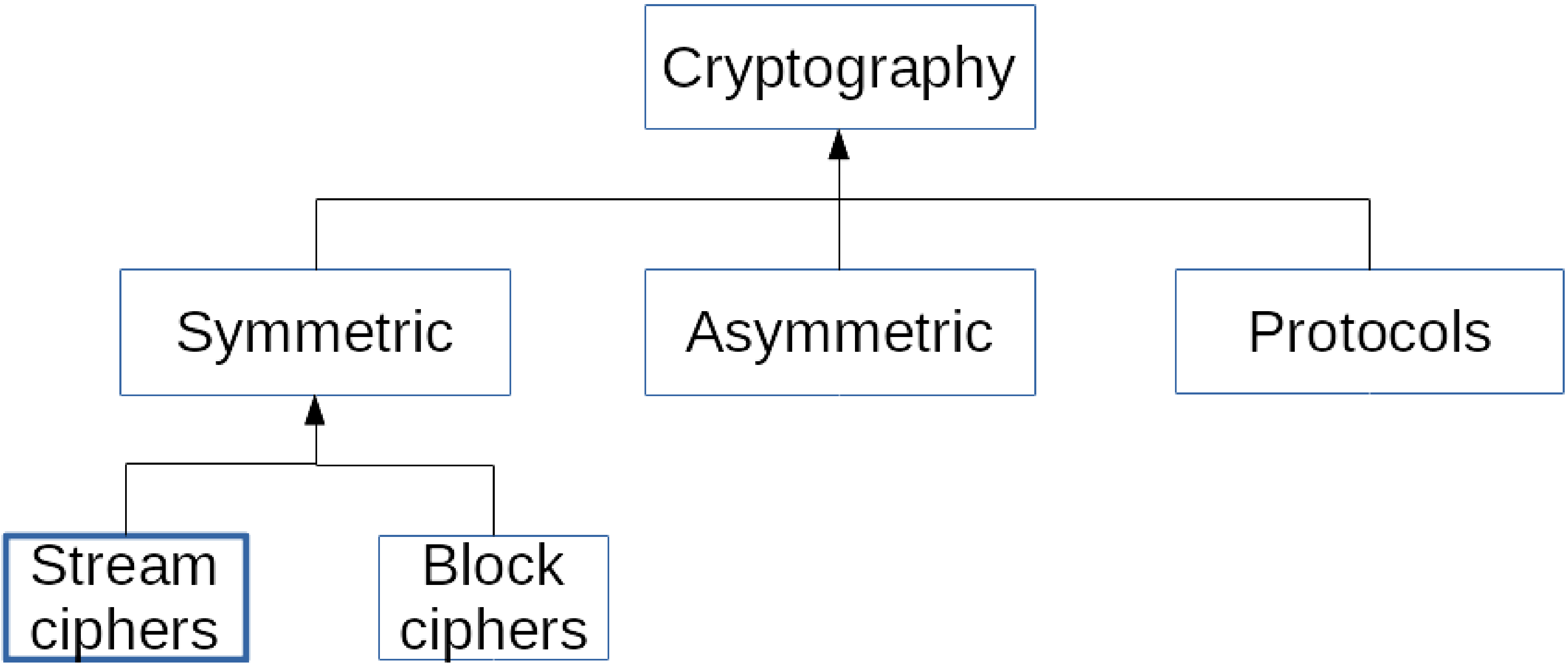
Last modified 2020 April 6

Topics

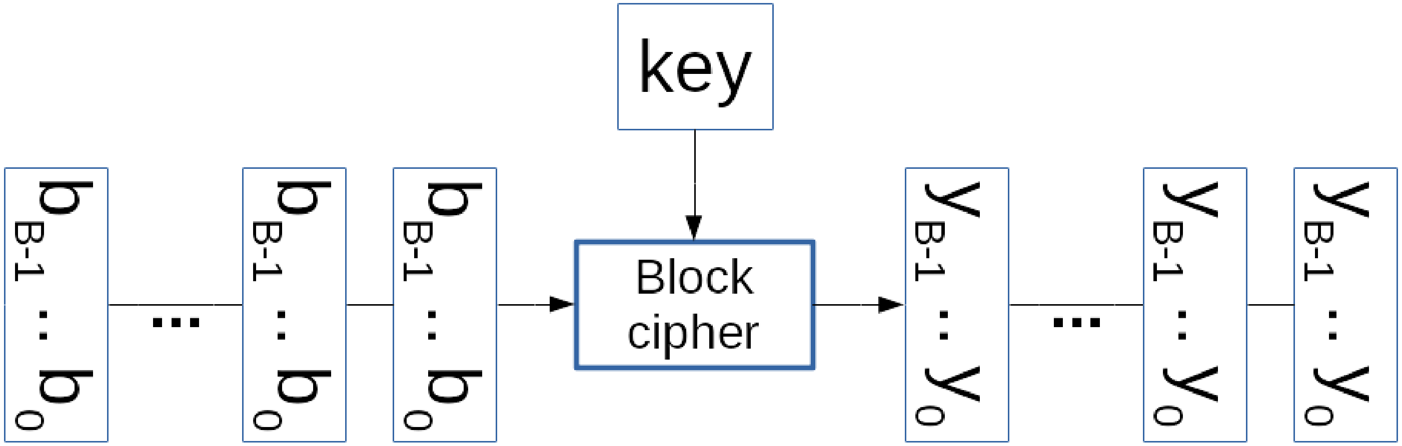
- Overview
- Block vs. Stream Ciphers
- XOR
- 3 Attempts at good streams
- Trivium

- Reading: "*Chapter 2: Stream Ciphers*" of Christof Paar and Jan Pelzl "*Understanding Cryptolography: A Textbook for Students and Practitioners*"

Overview

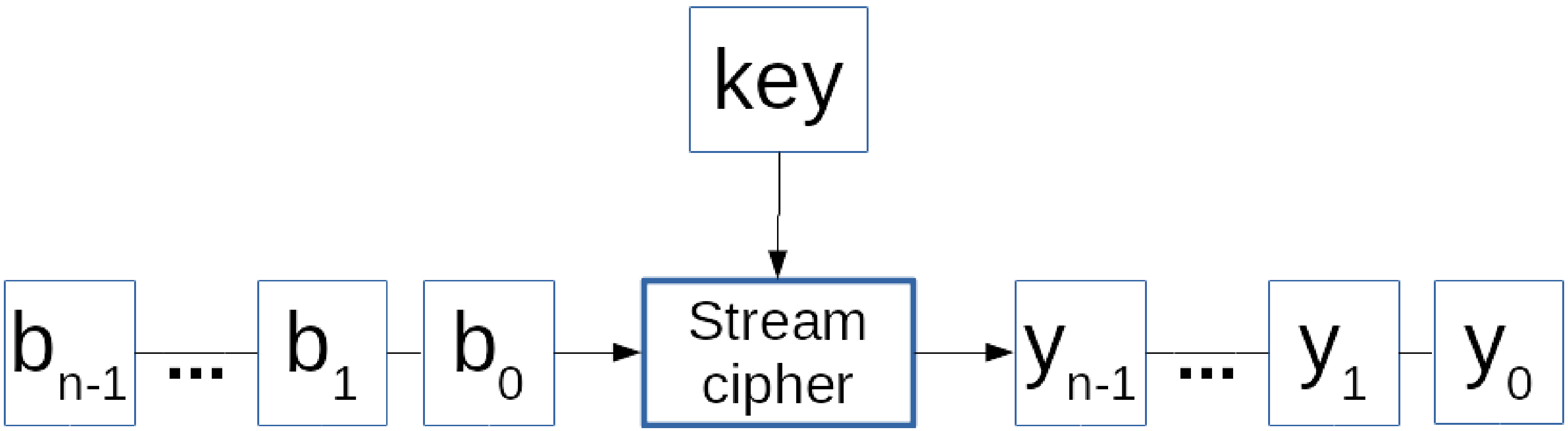

Cryptography
Symmetric
Asymmetric
Protocols
Stream ciphers
Block ciphers

Block cipher



- Encrypt blocks of B bits at a time
- Examples:
  - DES (block size = 64 bits = 8 bytes)
  - AES (block size = 128 bits = 16 bytes)

Stream Cipher



- Encrypt only ***<u>1</u>*** bit at a time
- Special case of Block cipher:
  - Blocksize B = 1
- Examples:
  - A5/1: part of GSM mobile phone standard for voice
  - RC4: some Internet traffic

Block vs. Stream

- Block
  - More popular (esp. for Internet)
- Stream
  - More efficient (esp. in hardware)

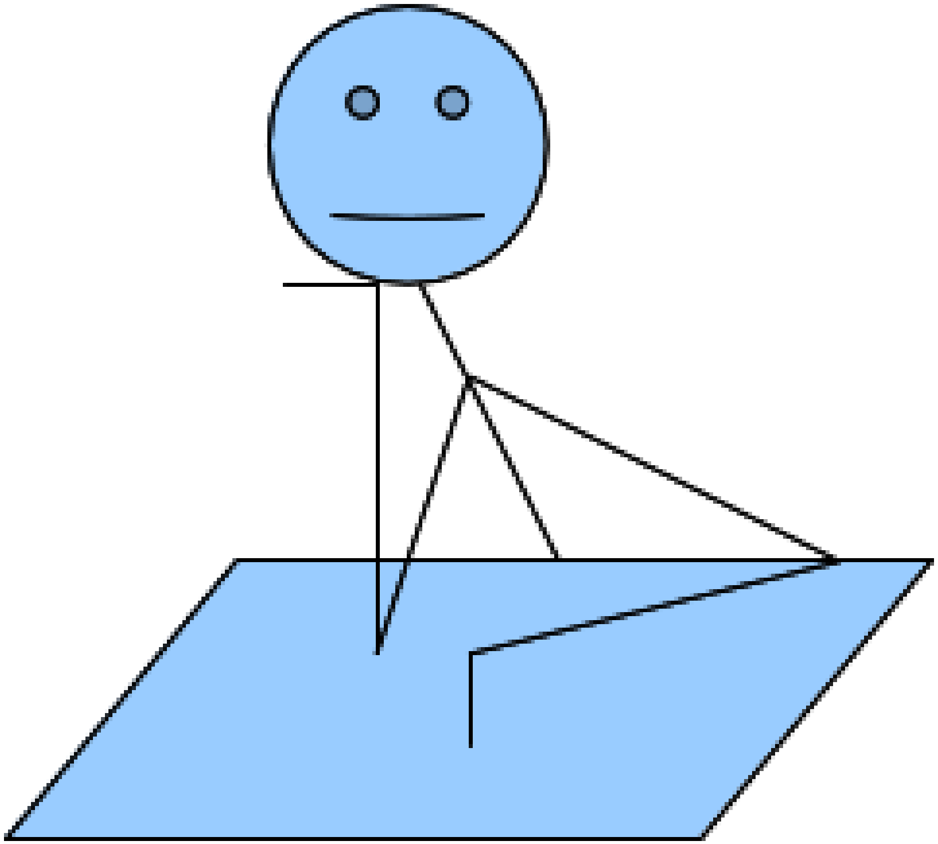
Gilbert Sandford Vernam

- Invented stream ciphers in 1917
- Patented by Vernam at ATT in 1919
- Later enhanced by Captain Joseph Mauborgne of the U.S. Army’s Signal Corps

Stream Ciphers
What happens inside?

- Let x_i, y_i, $s_i \in \{0, 1\}$
- Encrypt: $y_i = e_{si}() = x_i + s_i \bmod 2$
- Decrypt: $x_i = d_{si}() = y_i + s_i \bmod 2$
- NOTE:
  - Encrypt and decrypt with same function!

Curious student



"*So, what is that function?*"

XOR!

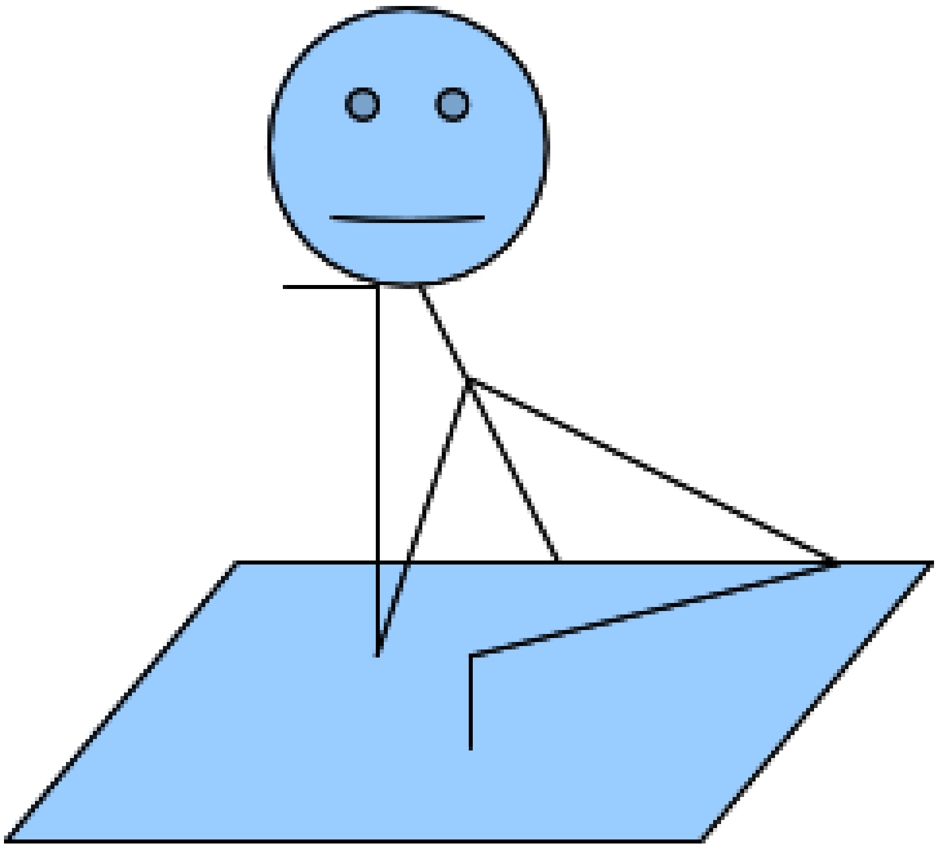
x	s	x + s mod 2	x XOR s
0	0	0+0 mod 2 = 0	0
0	1	0+1 mod 2 = 1	1
1	0	1+0 mod 2 = 1	1
1	1	1+1 mod 2 = 0	0

Intuition

- Unlike:
  - and, or, nand,nor
- XOR is
  - 0 half the time
  - 1 half the time

x	y	x and y	x or y	x nand y	x nor y
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Astute Student



"*Hey! If we always use the XOR function, then then Odious Oscar will know that!*

The key stream better be very secure!"

You are right!

- Attempt #1: One Time Pad
- Attempt #2: Ordinary Pseudo-random number generator
- Attempt #3: Cryptographic Pseudo-random number generator

One-time Pad

- Attempt #1: One Time Pad
- Record some truly events
  - Coin flipping
  - Semiconductor noise
  - Radioactive decay
- Advantage(s):
  - Truly unbreakable

One-time Pad: Disadvantages

- Hassle of physically transferring bits
  - Want to send 2 MByte image?
  - Need 2 MBytes of ***new*** bits
- Do not reuse that pad!
  - For a while during and after World War II, the UK and USA were able to read Soviet secrets because they re-used code books (http://www.crypto-it.net/eng/attacks/two-time-pad.html 2020 April 6)

Ordinary Pseudo-Random Number Generator

- Linear congruential generator
  - S_0 = seed
  - $S_{i+1} = A*S_i + B \bmod m$
  - The key is (S_0, A, B)
- Advantage
  - Pseudo-random number generators are readily available in many programming languages

Ordinary Pseudo-Random Number Generator: Disadvantage

- Easy to crack!
- Assume
  - S_0 is 128 bits
  - A, B are 64 bits each
  - 256 bit key total
- Also assume:
  - Oscar knows (or can guess) first 384 bits
  - E.g. standard header

Ordinary Pseudo-Random Number Generator: Disadvantage

1. Oscar gets key stream from cipher text and plain text guess:
   - $s_i \equiv x_i + y_i \bmod m$ ($0 \le i \le 383$)
2. Oscar splits s_i into 3 ranges
   - $S_0 = (s_0, .. s_{127})$
   - $S_1 = (s_{128}, .. s_{255})$
   - $S_2 = (s_{256}, .. s_{383})$
3. Oscar creates 2 equations:
   - $S_1 \equiv A*S_0 + B \bmod m$
   - $S_2 \equiv A*S_1 + B \bmod m$

Ordinary Pseudo-Random Number Generator: Disadvantage

4. Two equations, two unknowns, solve the equations!
   - $A \equiv (S_1 – S_2)/(S_0 – S_1) \bmod m$
   - $B \equiv S_1 - S_0*(S_1 – S_2)/(S_0 – S_1) \bmod m$
5. Actually get multiple solutions because $\gcd((S_0 – S_1),m) \neq 1$
   - But Oscar has a dramatically reduced search space

Cryptographic Pseudo-random number generator

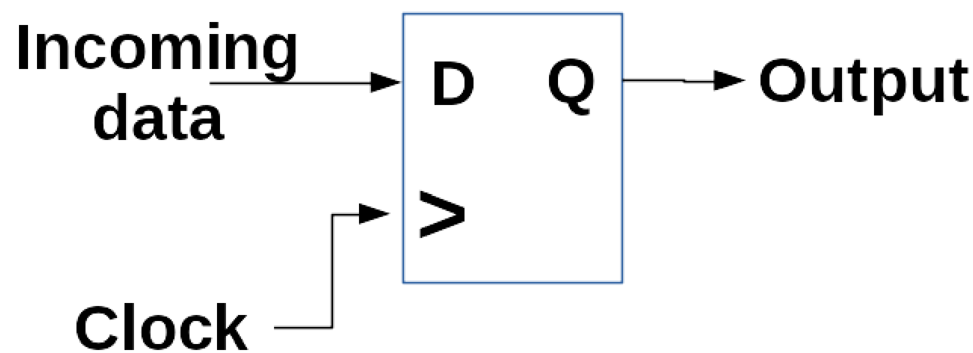
- Want seemingly contradictory requirements:
- Deterministic computability
  - So Bob can reproduce bit stream
- Seemingly random
  - So given a portion of bit stream, Oscar has a hard time figuring out what comes next

Linear Feedback Shift Registers and Flip-Flops

- We need to shift bits around as we want!
- We need flip-flops
- ***No! Not these!*** . . .

Linear Feedback Shift Registers and Flip-Flops

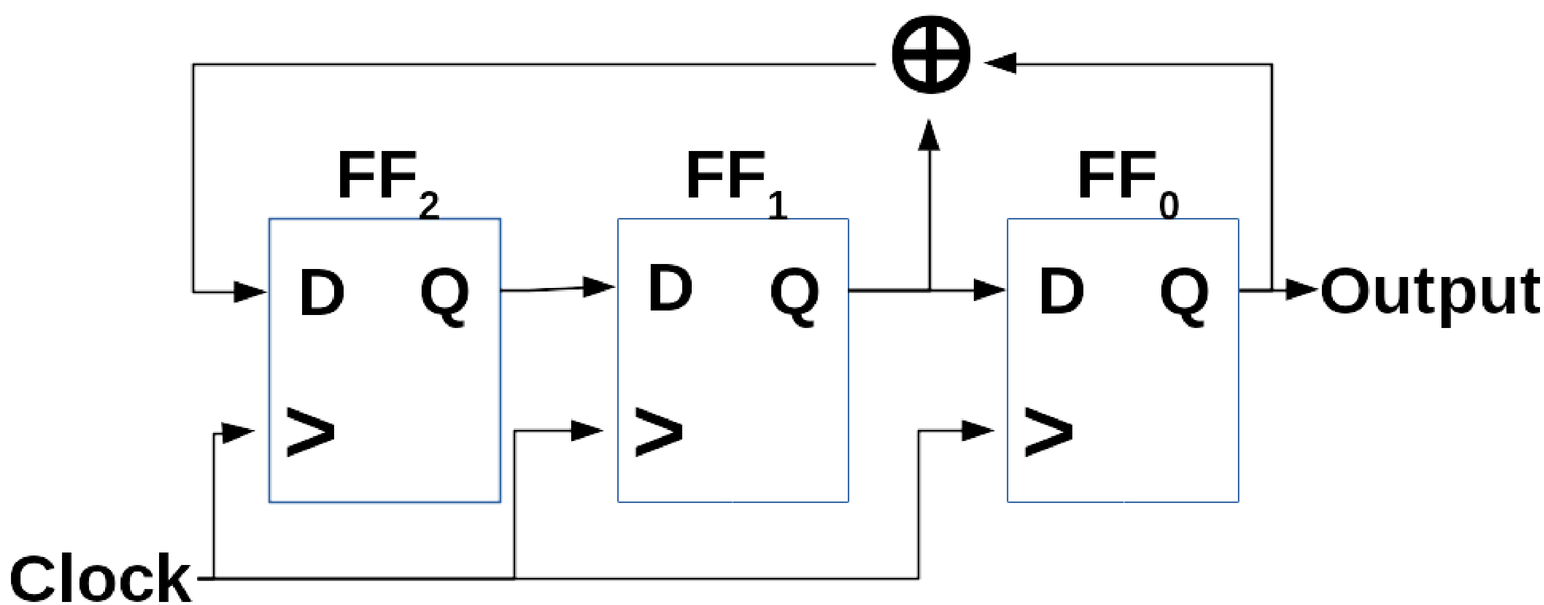
- Building block of digital circuits
- Holds and outputs one bit (Q)
- When clock cycles
  - (e.g. up and then down)
  - Gets new incoming bit (D)
- Holds and output new bit until clock cycles again



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

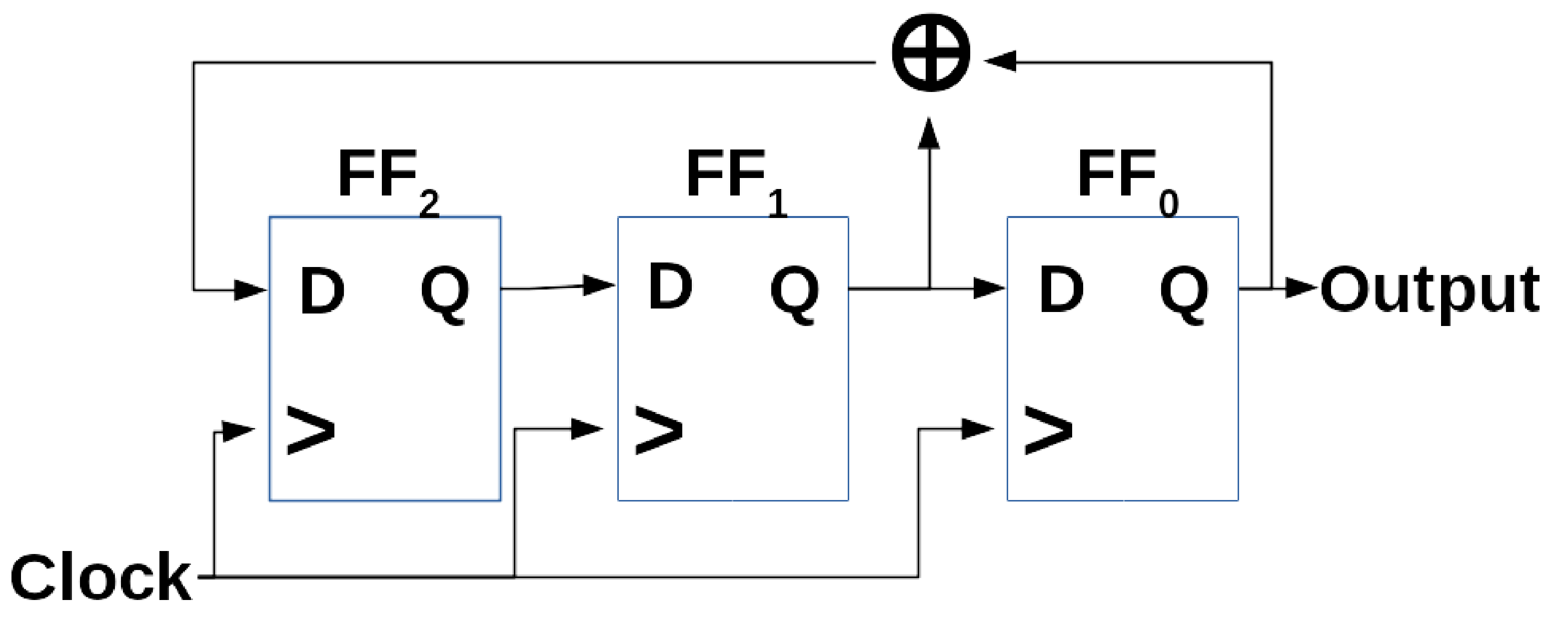
Clk	FF_2	FF_1	FF_0
0	1	0	0



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - ⊕ = XOR

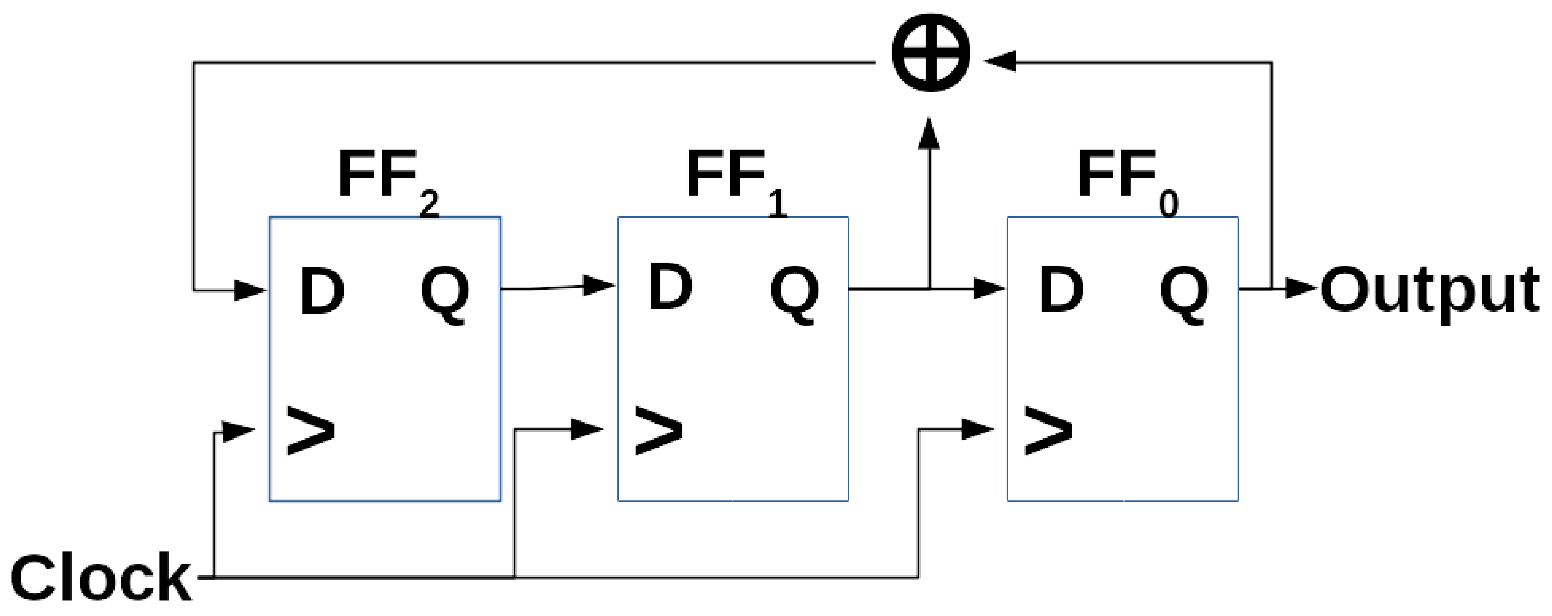
Clk	FF_2	FF_1	FF_0
0	1	0	0
1			?



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - ⊕ = XOR

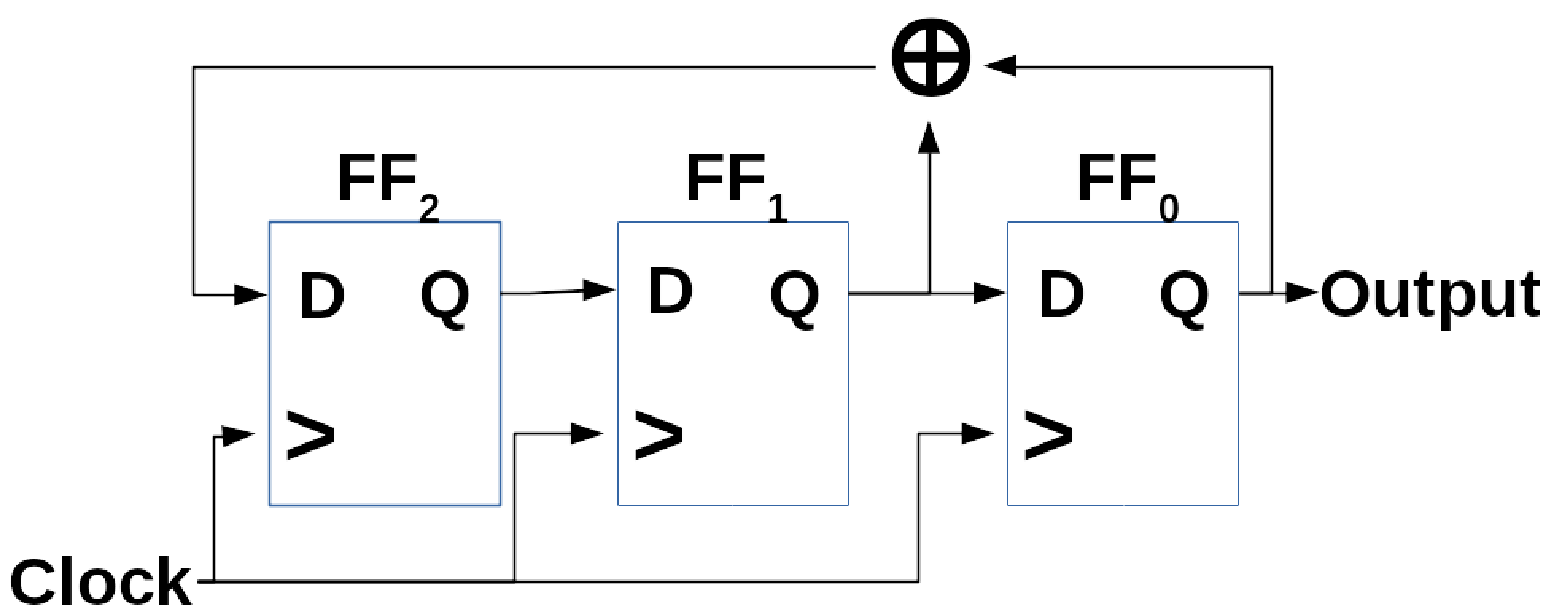
Clk	FF_2	FF_1	FF_0
0	1	0	0
1			0



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

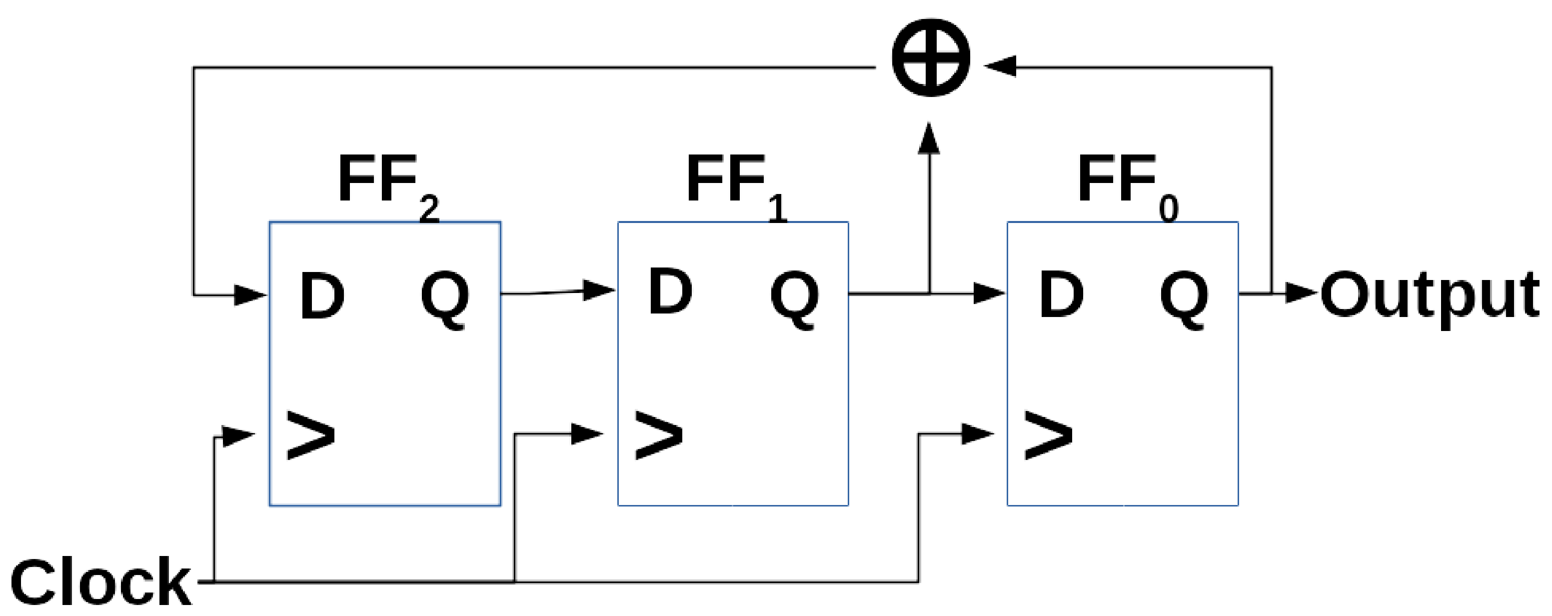
Clk	FF_2	FF_1	FF_0
0	1	0	0
1		?	0



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

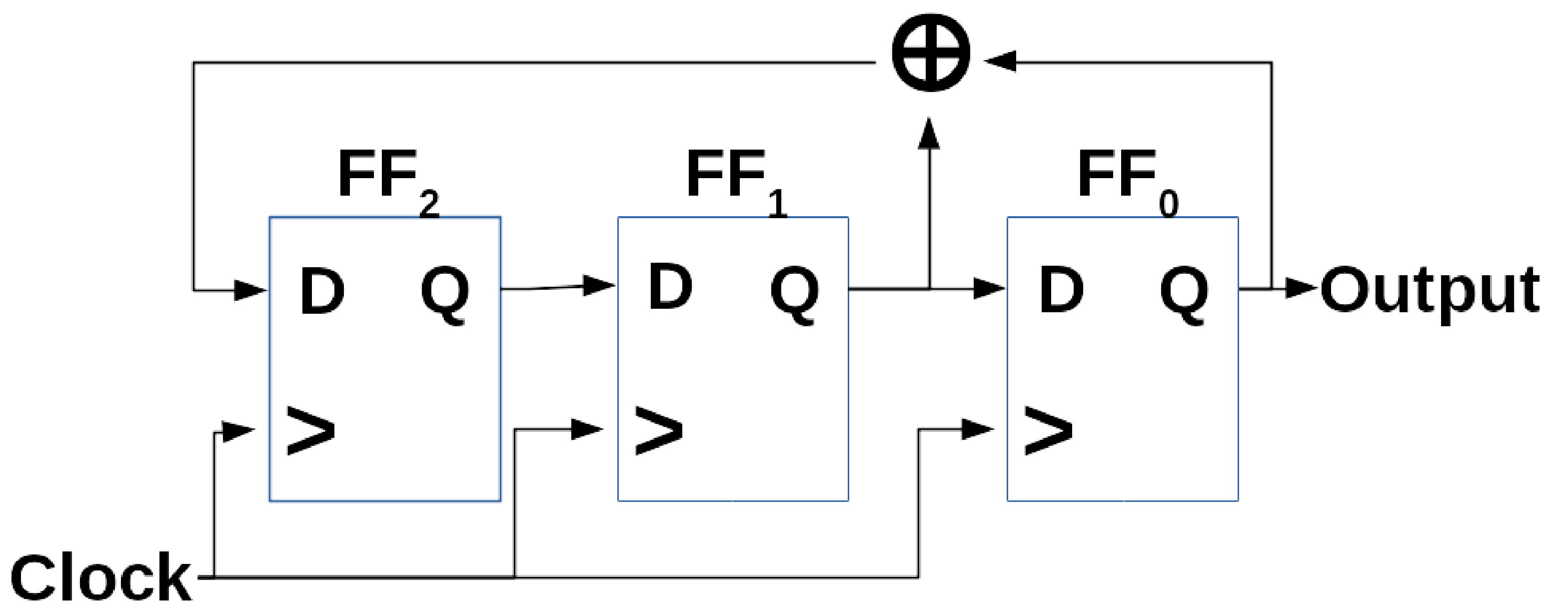
Clk	FF_2	FF_1	FF_0
0	1	0	0
1		1	0



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

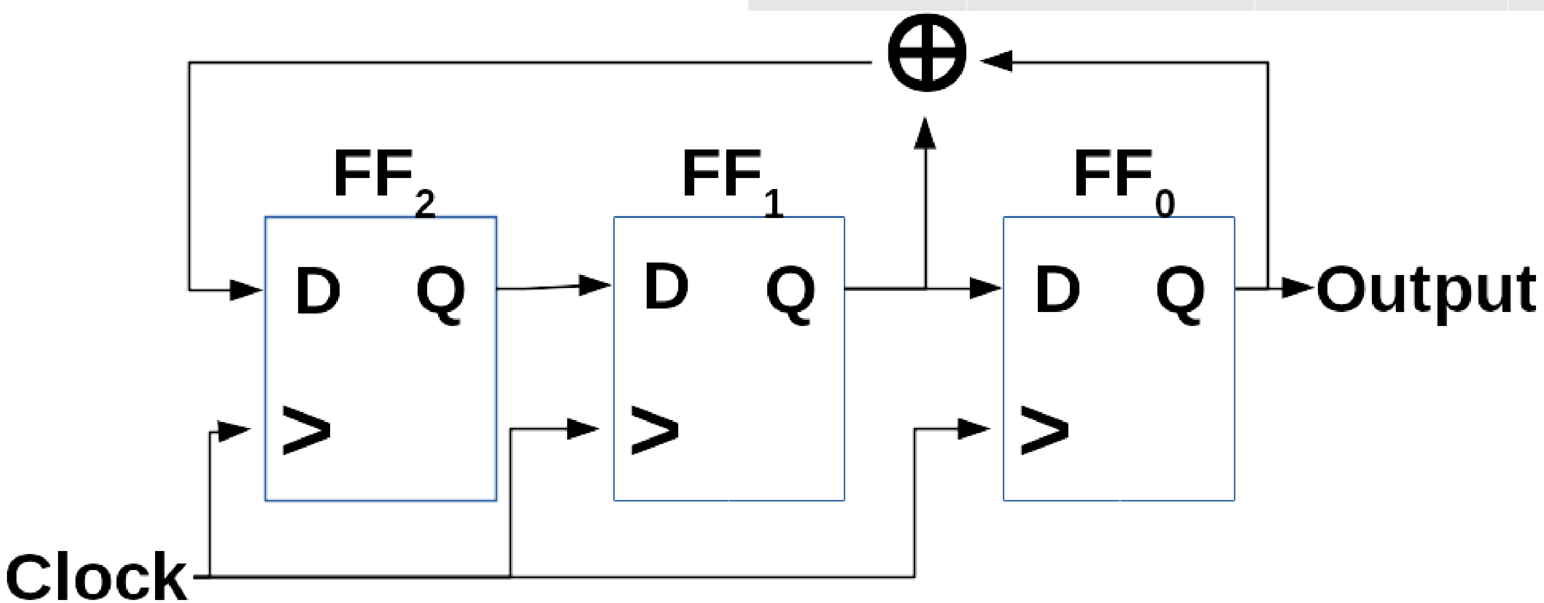
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	?	1	0



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

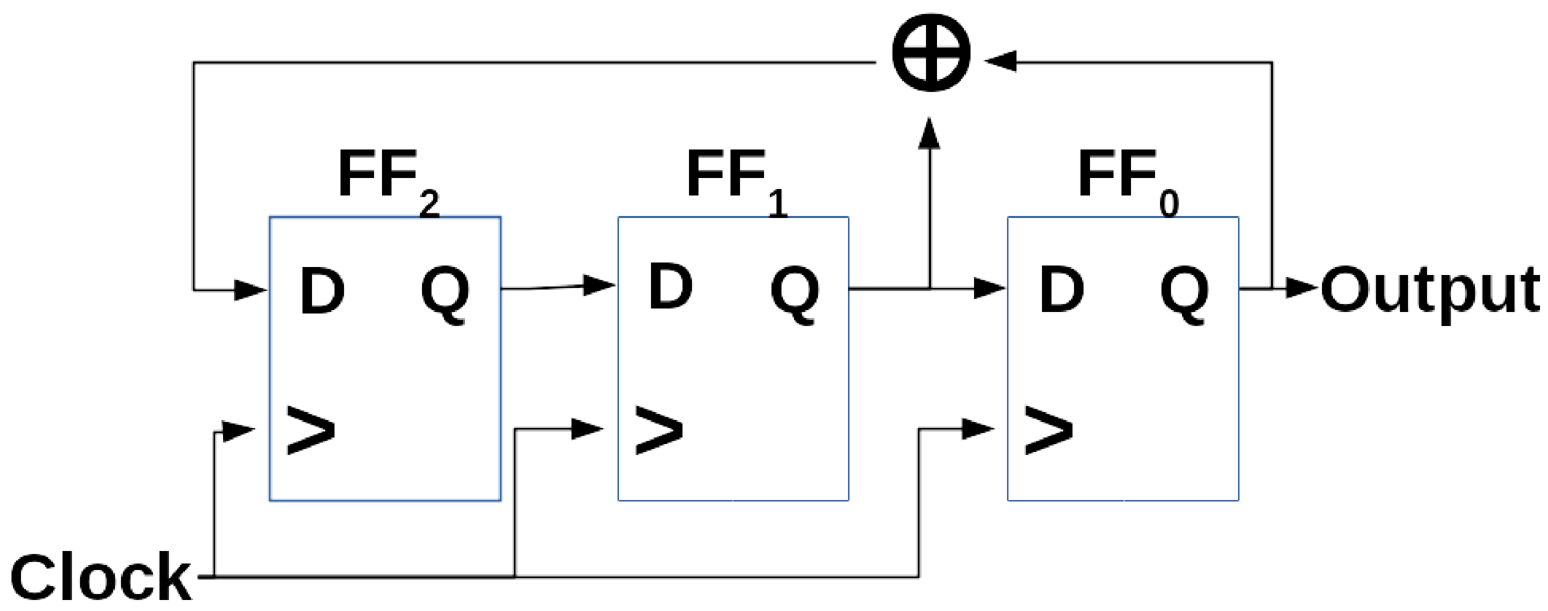
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	$0 \oplus 0 = 0$	1	0



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - ⊕ = XOR

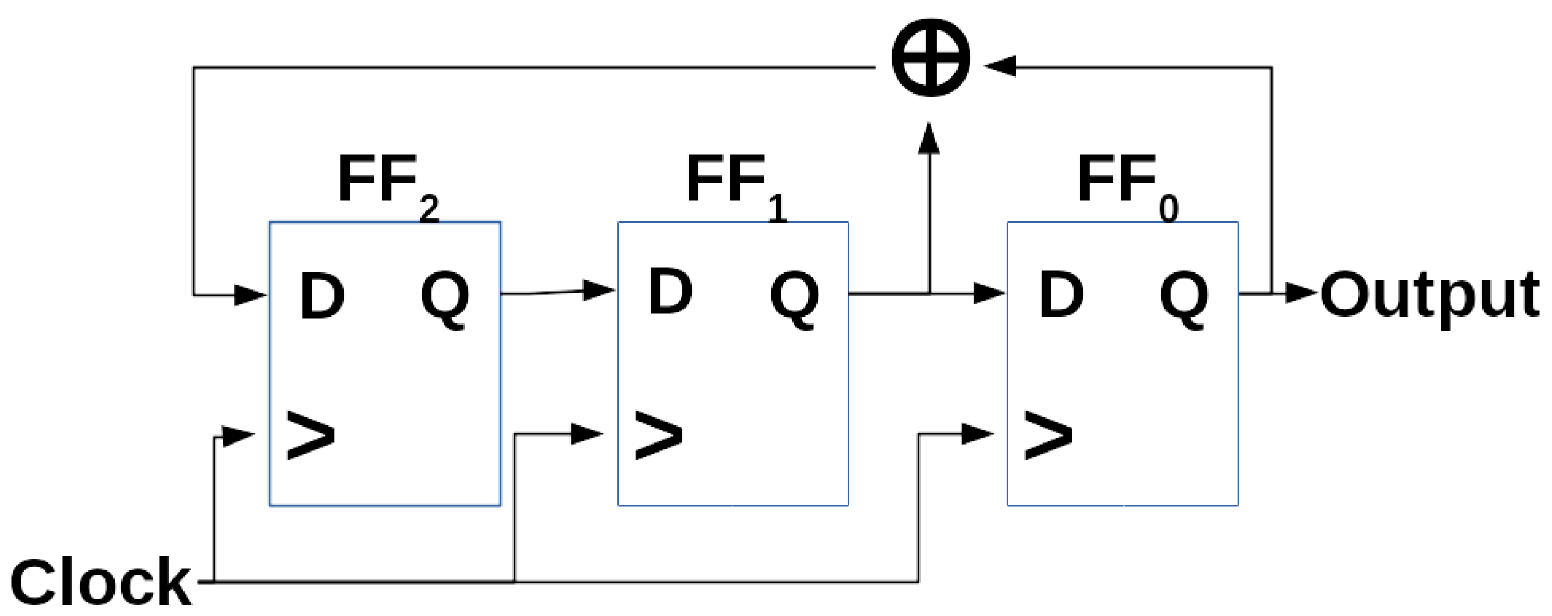
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	0	1	0
2			?



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

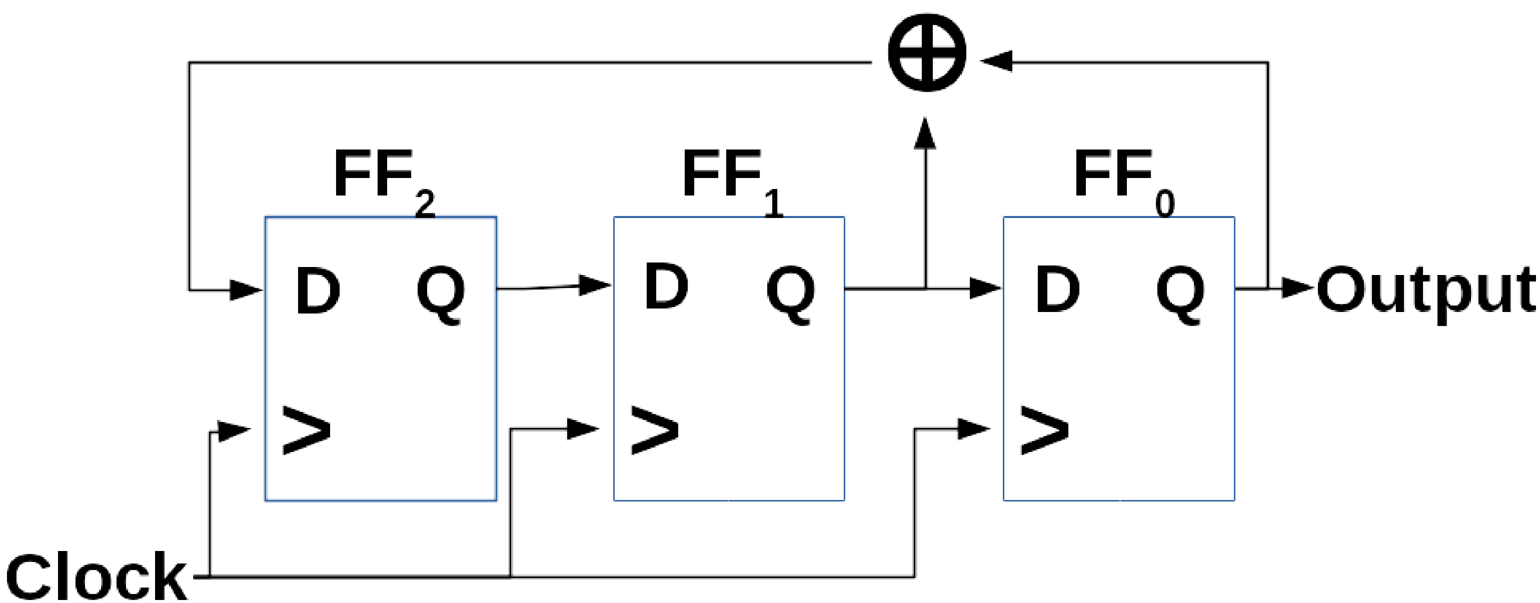
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	0	1	0
2			1



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - ⊕ = XOR

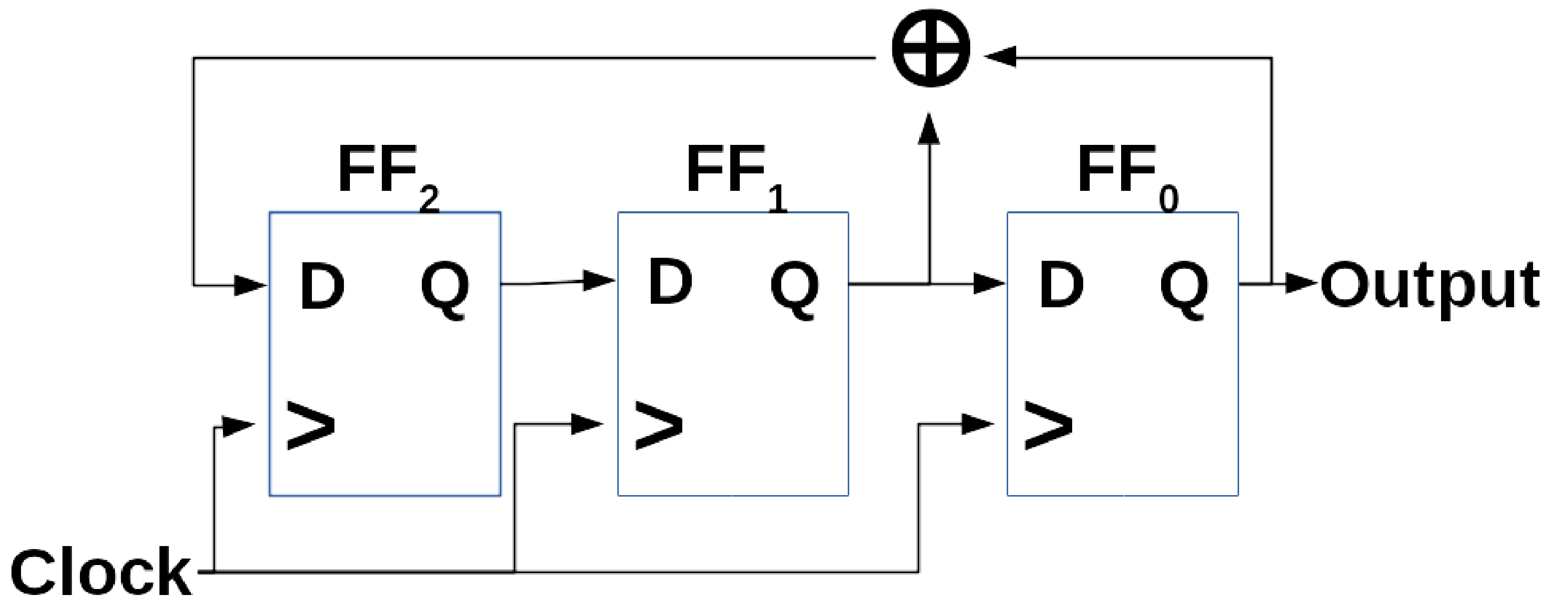
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	0	1	0
2		?	1



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - ⊕ = XOR

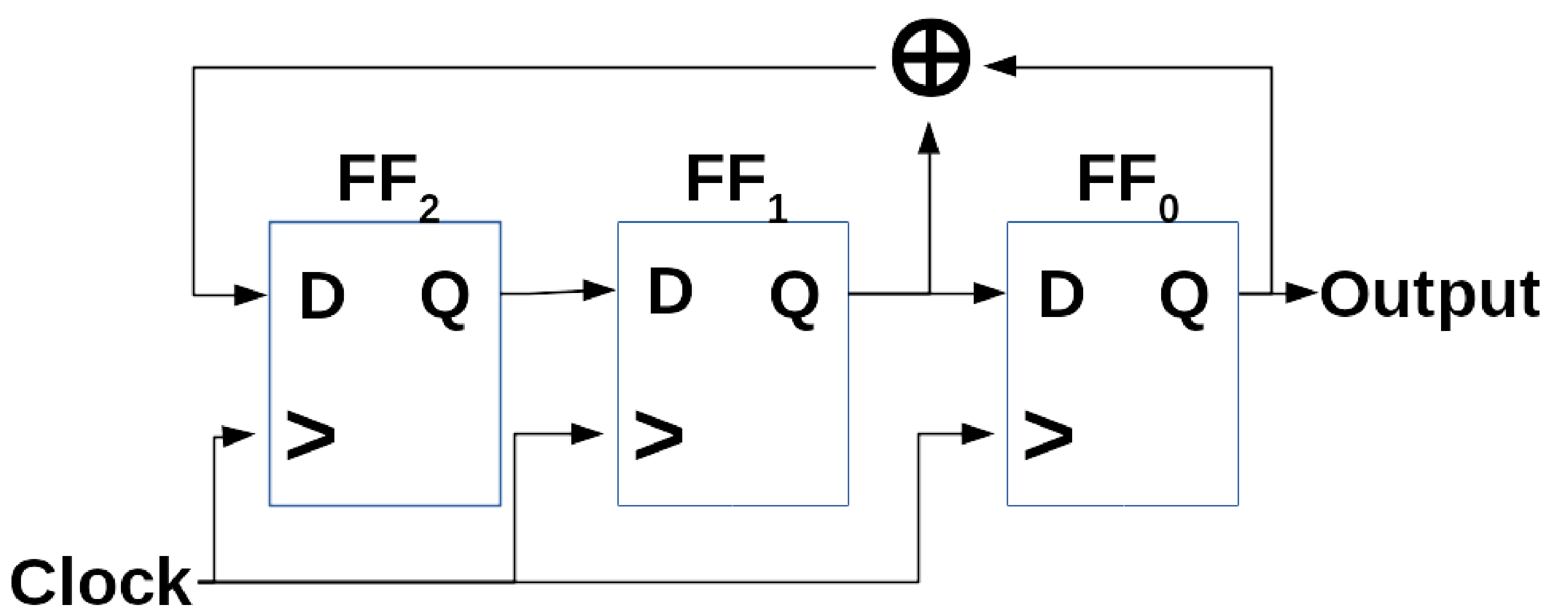
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	0	1	0
2		0	1



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

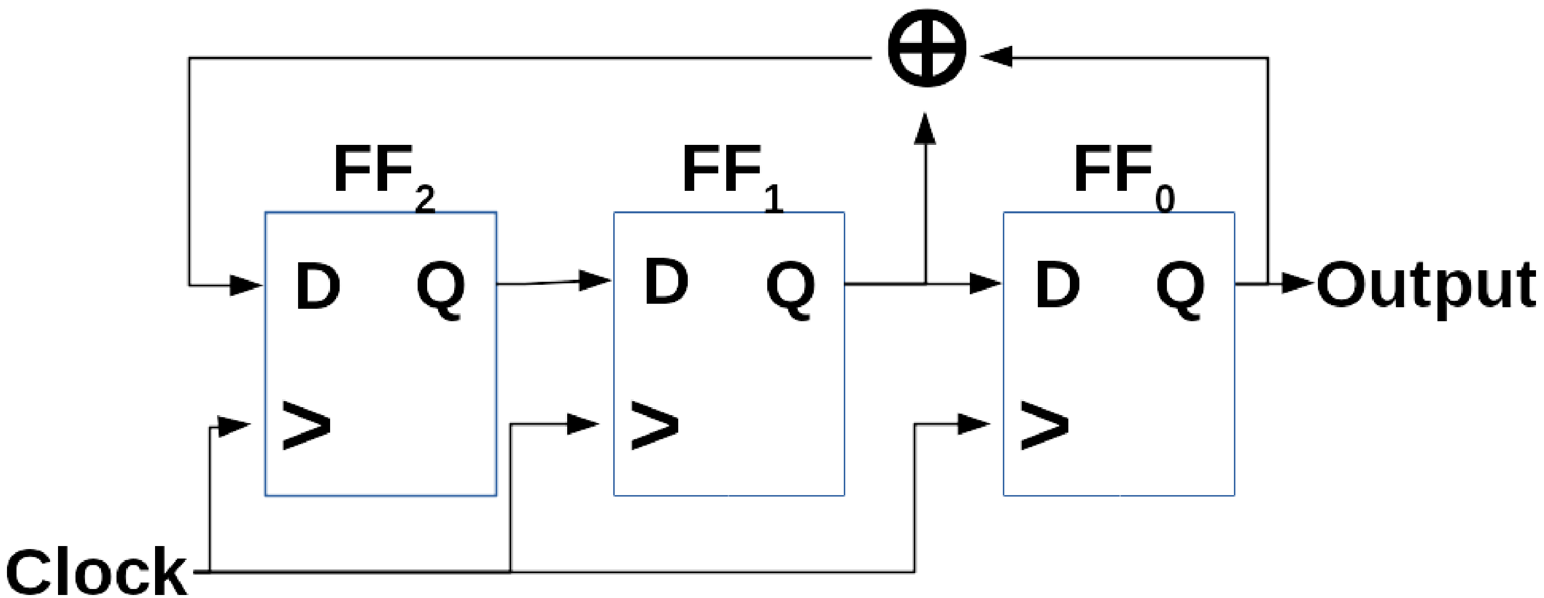
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	0	1	0
2	?	0	1



Linear Feedback Shift Registers

- Example with 3 flip-flops:
  - $\oplus$ = XOR

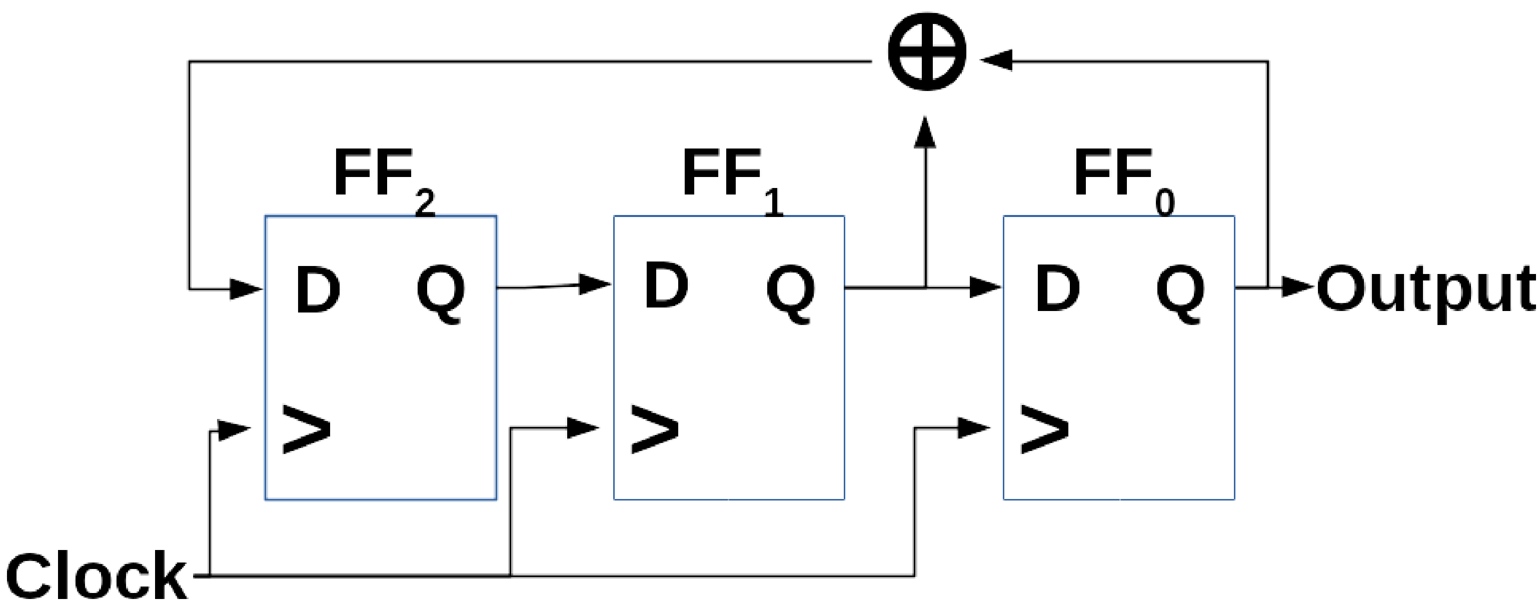
Clk	FF_2	FF_1	FF_0
0	1	0	0
1	0	1	0
2	$1 \oplus 0 = 1$	0	1



Linear Feedback Shift Registers

Clk	FF_2	FF_1	FF_0
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0

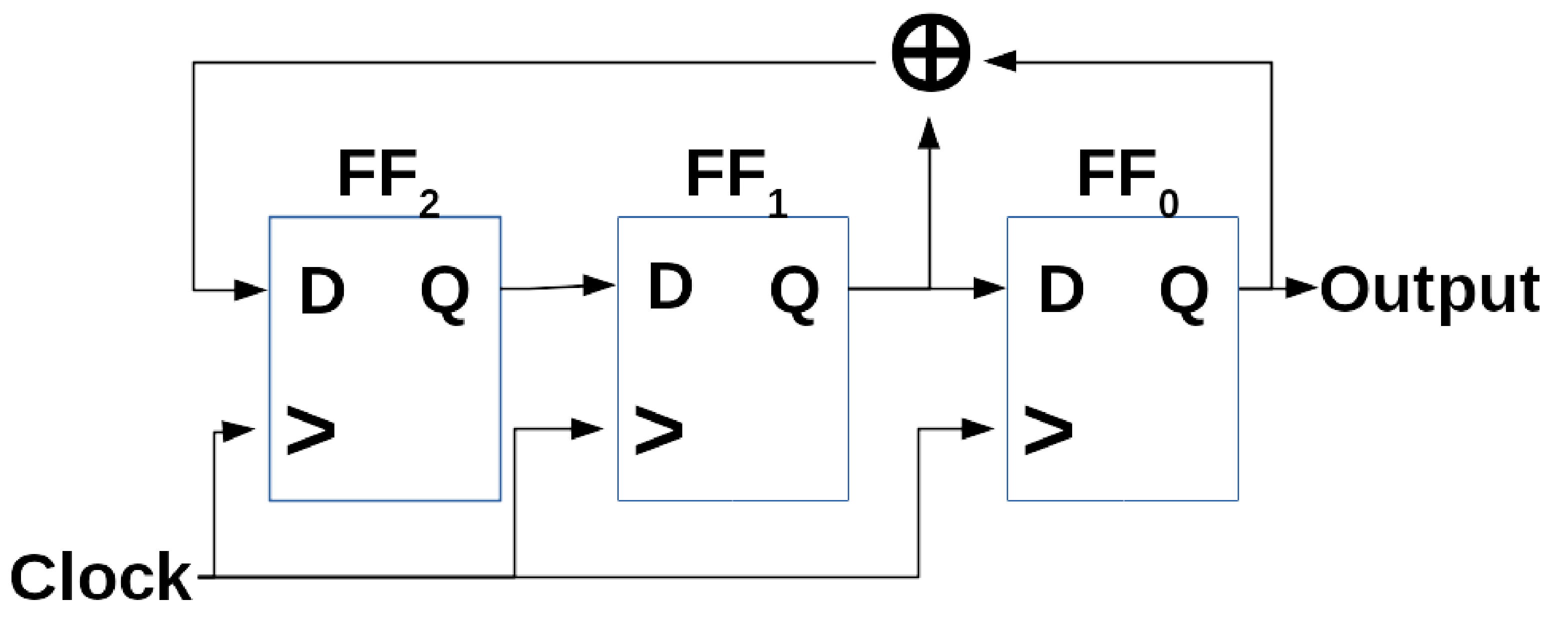
Clk	FF_2	FF_1	FF_0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0



Linear Feedback Shift Registers

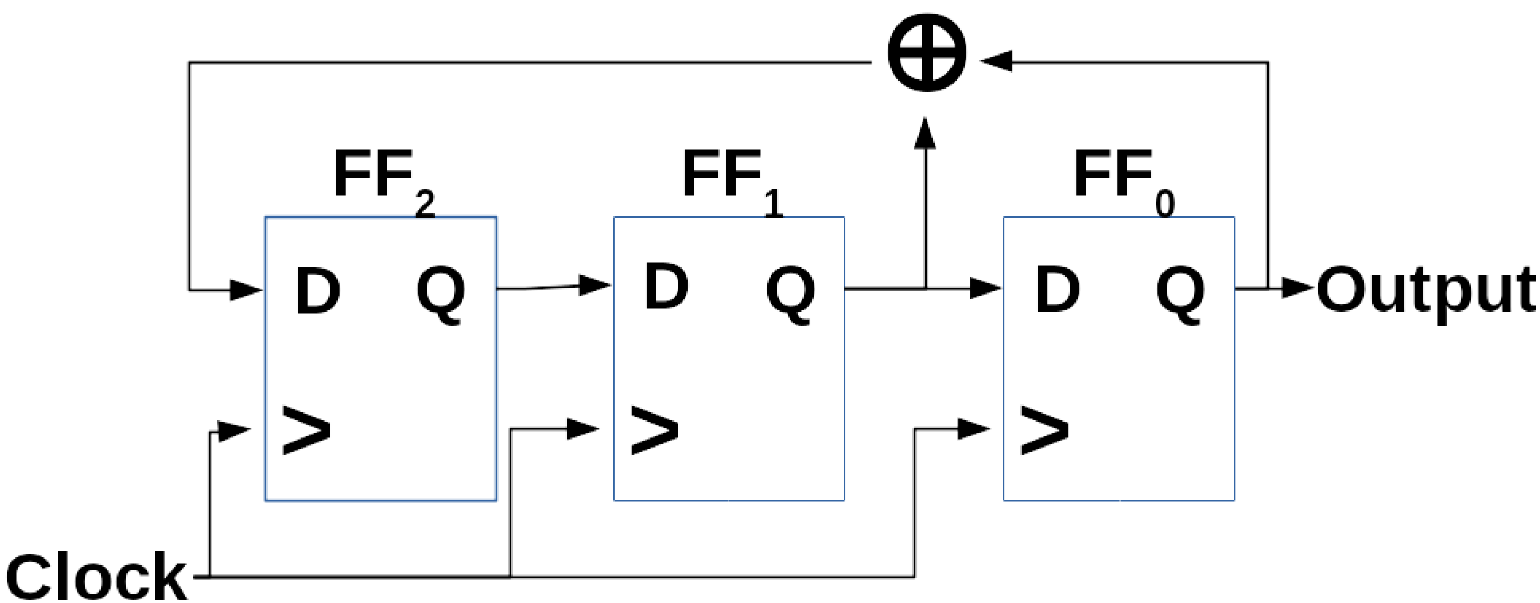
Clk	FF2	FF1	FF0
0	1	0	0
...	...	...	...
7	1	0	0

- ***Ruh-roh! It repeats!***



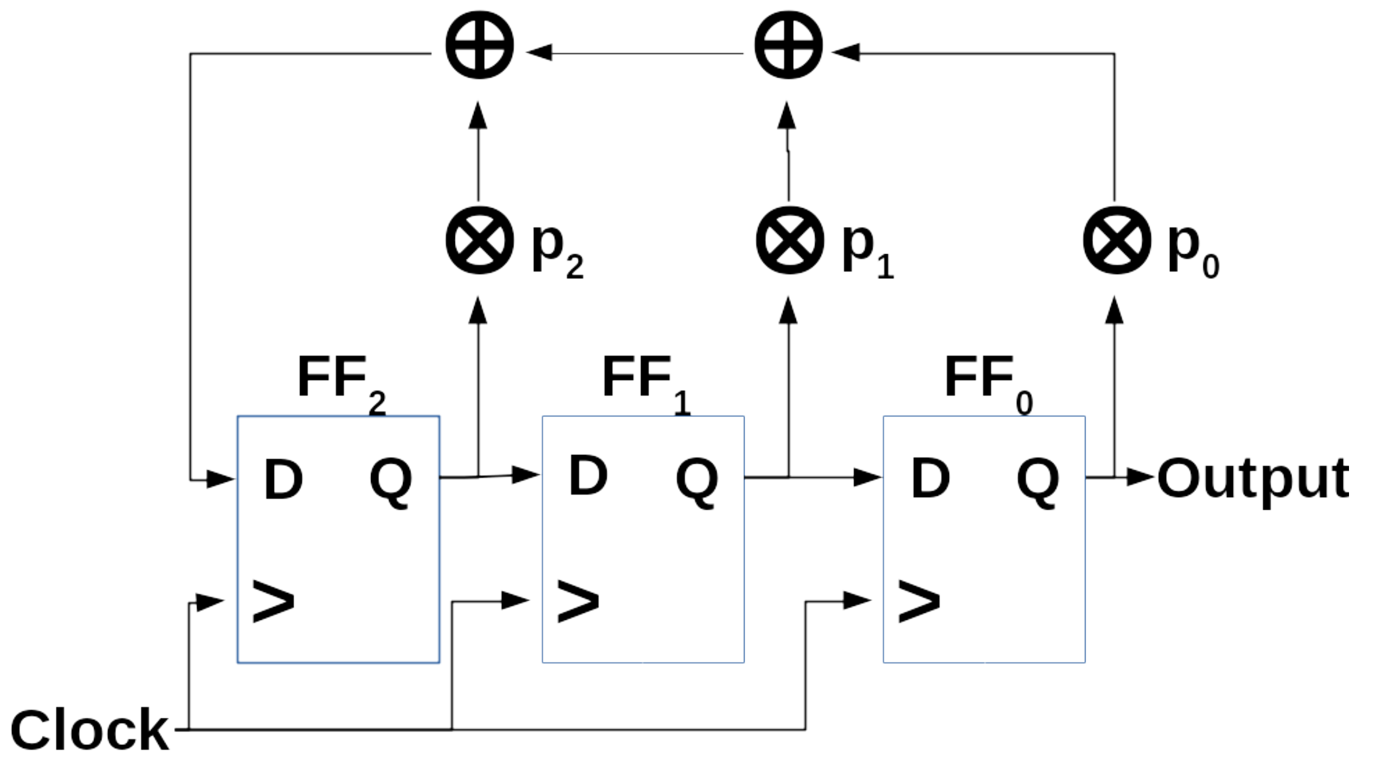
Linear Feedback Shift Registers

- It ***has to*** repeat
  - 3 flip-flops
  - 2 states per flip-flop (0 or 1)
  - 2^3 = 8 possible states total



Linear Feedback Shift Registers

- General form
  - (p_{n-1}, .. p_0) are switches
  - output(x) = $x^m + p_{m-1} * x^{m-1} + \ldots + p_1 * x + p_0$



Attack against Known-Plaintext LFSRs

- N flip-flops:
  - 2^N possible states
  - 2^N maximal cycle length

Attack against Known-Plaintext LFSRs

- Encryption details:
  - m digits
  - Secret key vector: p_{m-1}, p_{m-2}, … p_1, p_0
- Assume Oscar knows:
  - Value of m
  - Plain text: x_0, x_1, … x_{2m-1}
  - Cipher text: y_0, y_1, … y_{2m-1}

Attack against Known-Plaintext LFSRs

1. Oscar reconstructs 2m key stream bits
   - $s_i \equiv x_i + y_i$ mod 2; i = 0, 1, … 2m-1
2. Apply definition of resulting bit s_m from previous bits s_0, s_{m-1} and key vector p_0, p_{m-1}:
   - $s_{i+m} \equiv$ sum(j=0, j<= m-1, p_j *s_{i+j} mod 2);
   - s_i, $p_i \in \{0,1\}$
   - i =0, 1, ...
3. Generate m equations with m unknowns each:
   - i = 0; $s_m \equiv p_{m-1}*s_{m-1} + \ldots + p_1*s_1 + p_0*s_0$
   - i = 1; $s_{m+1} \equiv p_{m-1}*s_m + \ldots + p_1*s_2 + p_0*s_1$
   - . . .
   - i = m-1; $s_{2m-1} \equiv p_{m-1}*s_{2m-2} + \ldots + p_1*s_m + p_0*s_{m-1}$
4. Solve system of equations by linear algebra

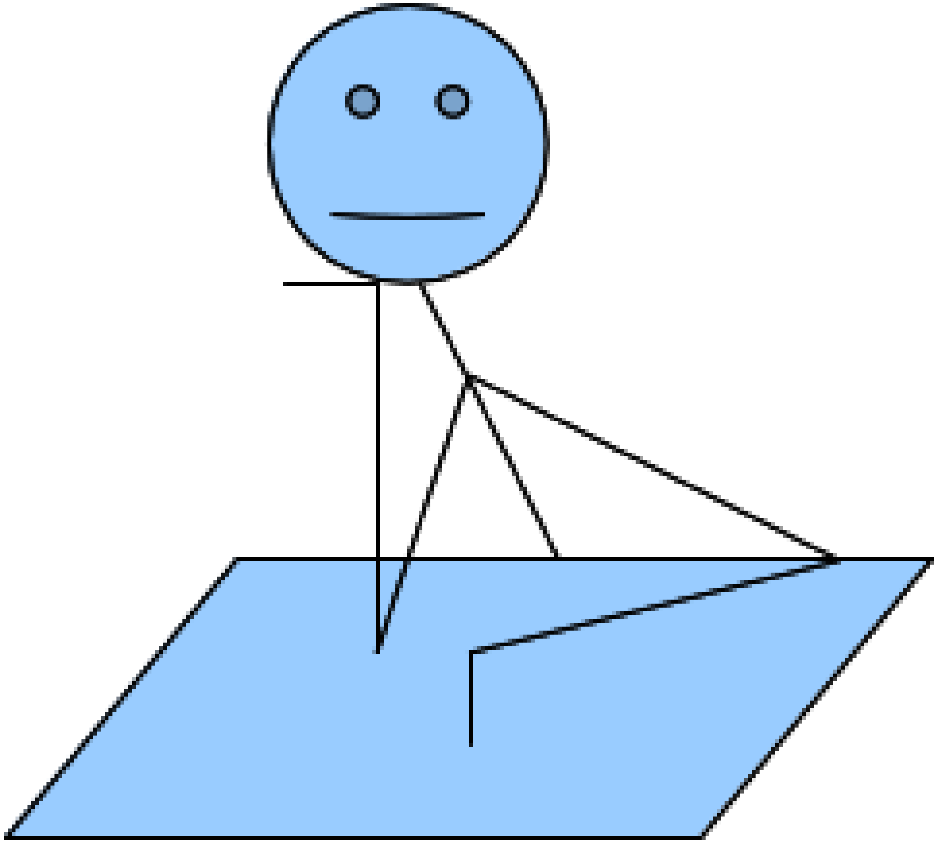
Astute Student

“***Hmm, it was solved with linear algebra!***

What if we:

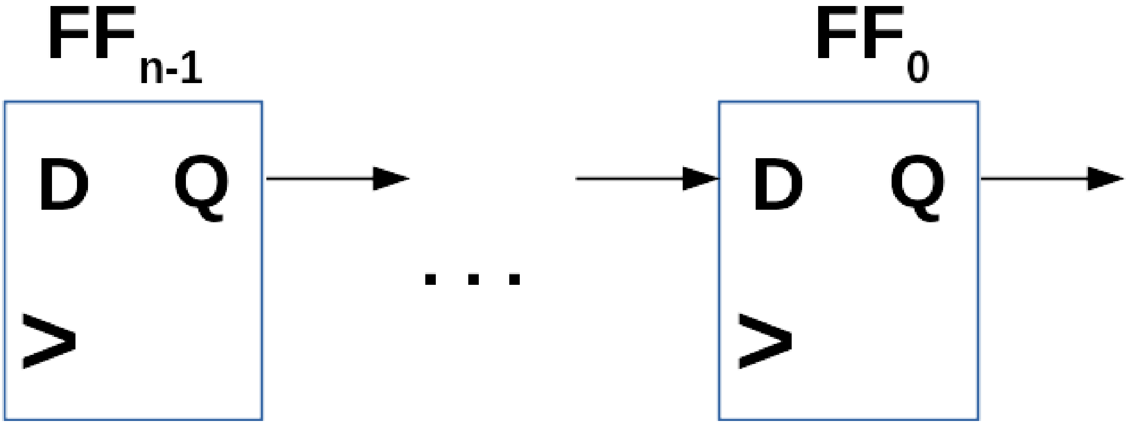
(1) add more state

(2) make it non-linear?”



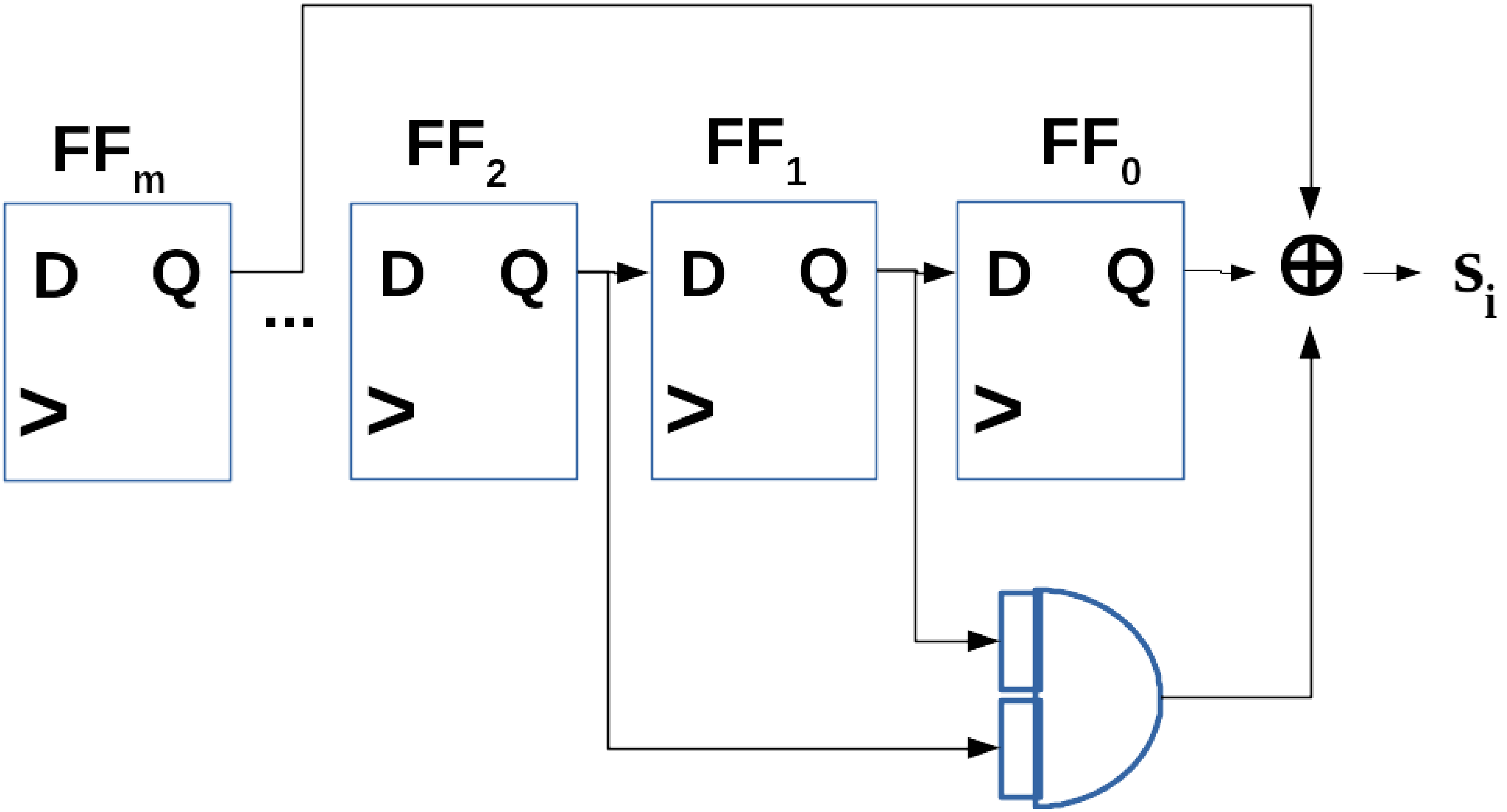
Add more state

- Shift a bit into long register
- n clock-cycles later, it shifts out

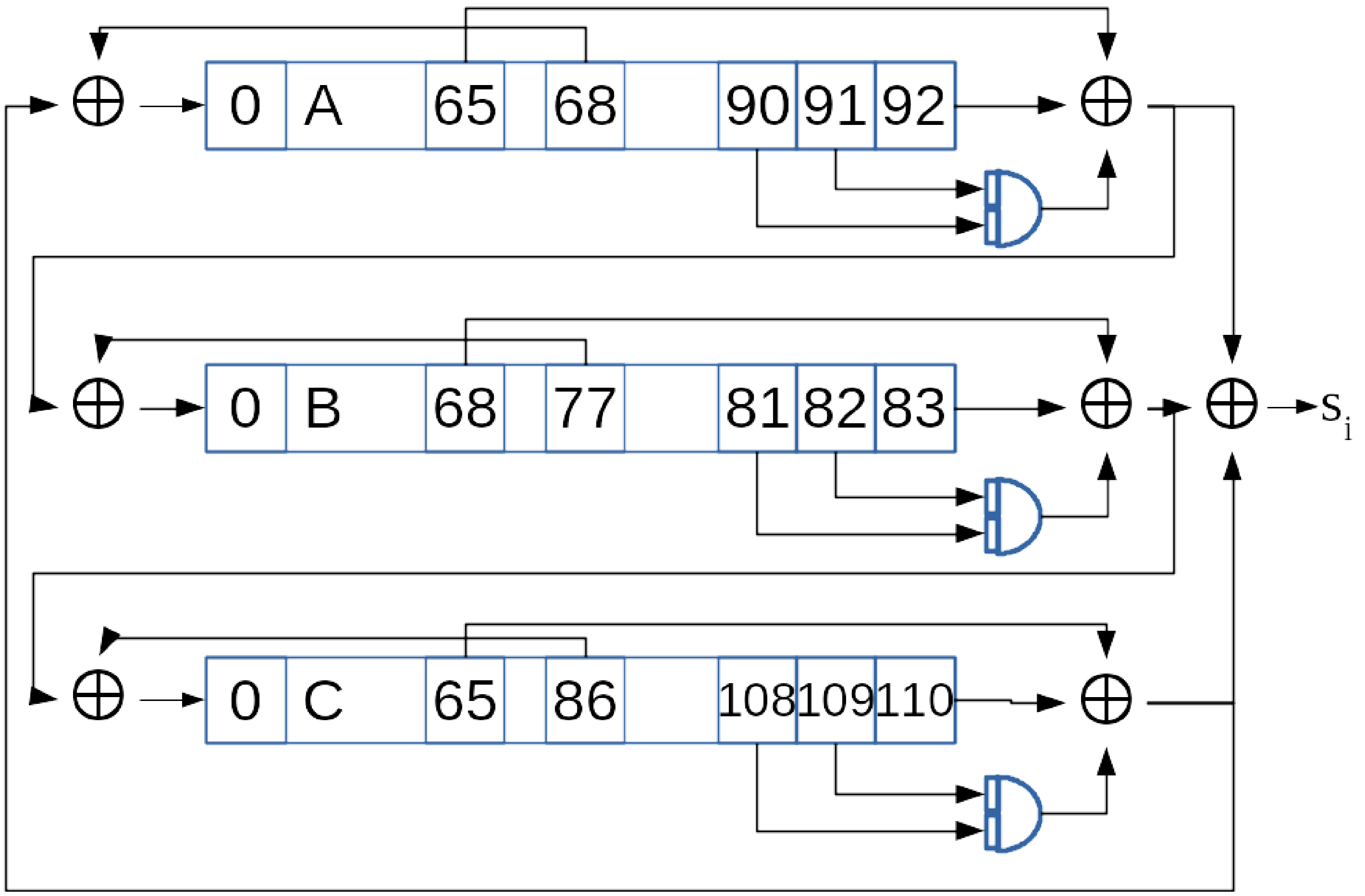


Make non-linear

- Output depends on more than just last flip-flop
  - This means "bitwise-AND" 
  - Bitwise And is multiplication: no longer linear



Trivium



Trivium: How to use

1. Need 80 bit key
   - Keep secret!
   - Load into Register A
2. Need 80 initialization vector
   - No need to keep secret, but must change between sessions (***Nonce*** = ***N***umber use ***ONCE***)
   - Load into Register B
3. Load last 3 bits of C with 1
4. Clear all other bits to 0
5. Run 4*(93+84+111) = 1152 times
   - Throw these away
6. Now use starting at 1153!

References:

- “*Chapter 2: Stream Ciphers*” of Christof Paar and Jan Pelzl “*Understanding Cryptolography: A Textbook for Students and Practitioners*”
- https://cryptologicfoundation.org/what-we-do/educate/bytes/this_day_in_history_calendar.html/event/2020/02/07/1581051600/1960-inventor-gilbert-vernam-died- (Downloaded 2020 April 6)
- http://www.crypto-it.net/eng/attacks/two-time-pad.html (Downloaded 2020 April 6)