

ORACLE

# LXM: Better Splittable Pseudorandom Number Generators (and Almost as Fast)

---

**Guy L. Steele Jr.**




Oracle Labs

**Sebastiano Vigna**

Università degli Studi di Milano

**ACM OOPSLA Conference**

**October 22, 2021**



Copyright © 2021 Oracle and/or its affiliates (“Oracle”). All rights are reserved by Oracle except as expressly stated as follows. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific written permission of Oracle.

# What Do We Want From a Pseudorandom Number Generator (PRNG)?

Many decades ago:

- A stream of floating-point values drawn uniformly from  $[0.0, 1.0)$ , approximated by drawing uniformly from the set  $\{ k/p \mid 0 \leq k < p \}$
- It was considered okay if the low-order bits were “not very random”

Now:

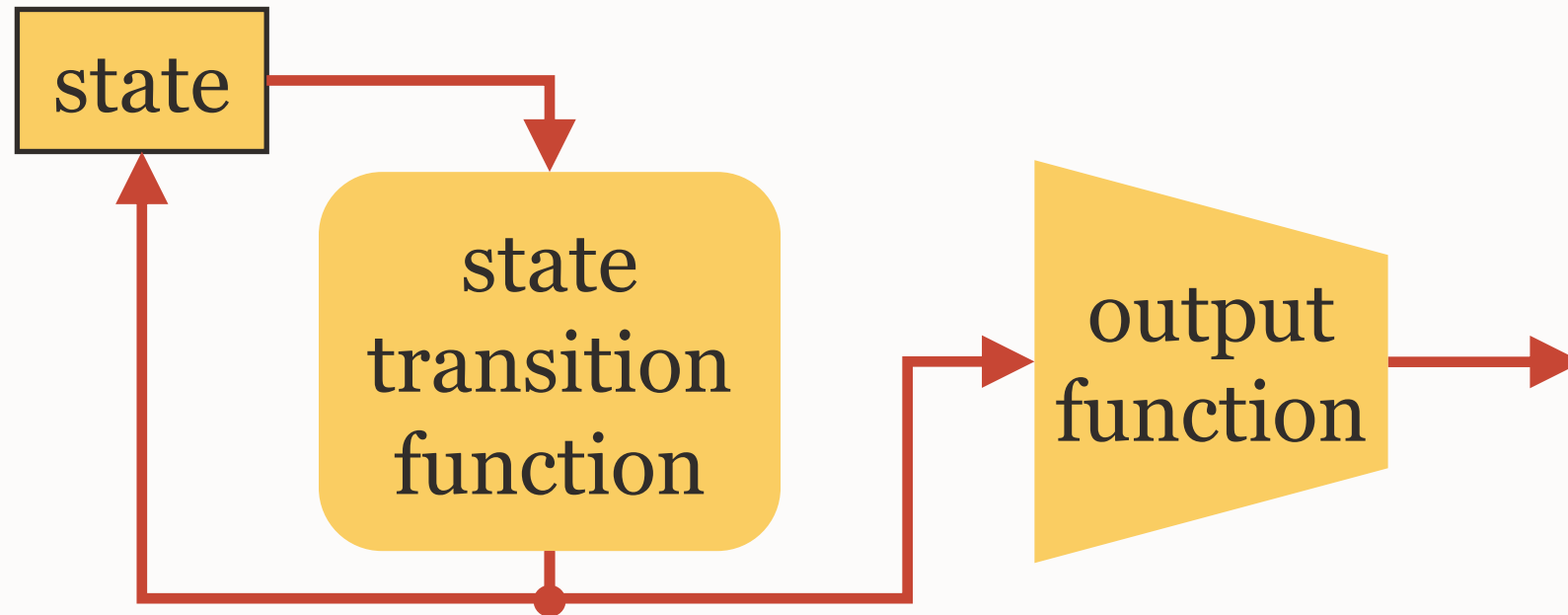
- A stream of floating-point values drawn uniformly from  $[0.0, 1.0)$   
(as above, but we expect  $p$  to be much, much larger)  
**or** a stream of  $w$ -bit integers drawn uniformly from  $[0, 2^w)$   
(we expect all bits of each integer to be “equally random”)

# What Do We Want From a Pseudorandom Number Generator (PRNG)?

What else has changed?

- Moore's Law: computers are much faster now
  - Applications can draw many more numbers
    - A PRNG that repeats its sequence after  $2^{32}$  values is *not* okay
    - In fact, repeating after  $2^{64}$  values is not that great
  - PRNG test suites are much more discriminating
    - We now routinely test *trillions* of generated values, rather than millions, looking for subtle statistical anomalies
- Parallelism (either SIMD or multithreading)
  - Not just one generator: dozens, or millions

# Basic Structure of a Pseudorandom Number Generator (PRNG)

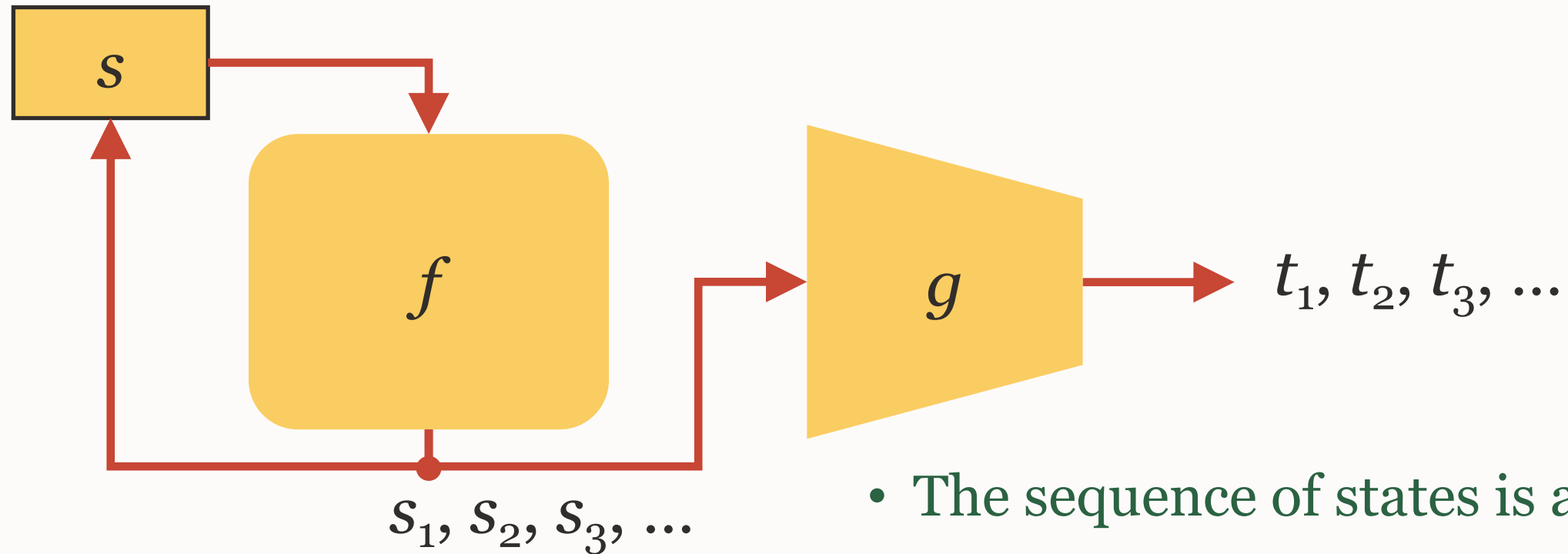


## Mathematical Description

initial state  $s_0$

$$s_k = f(s_{k-1})$$

$$t_k = g(s_k)$$



- The sequence of states is a *cycle*.
- Smallest  $k$  for which  $s_k = s_0$  is called the *period* of the generator.

- We assume  $f$  is *bijective*.

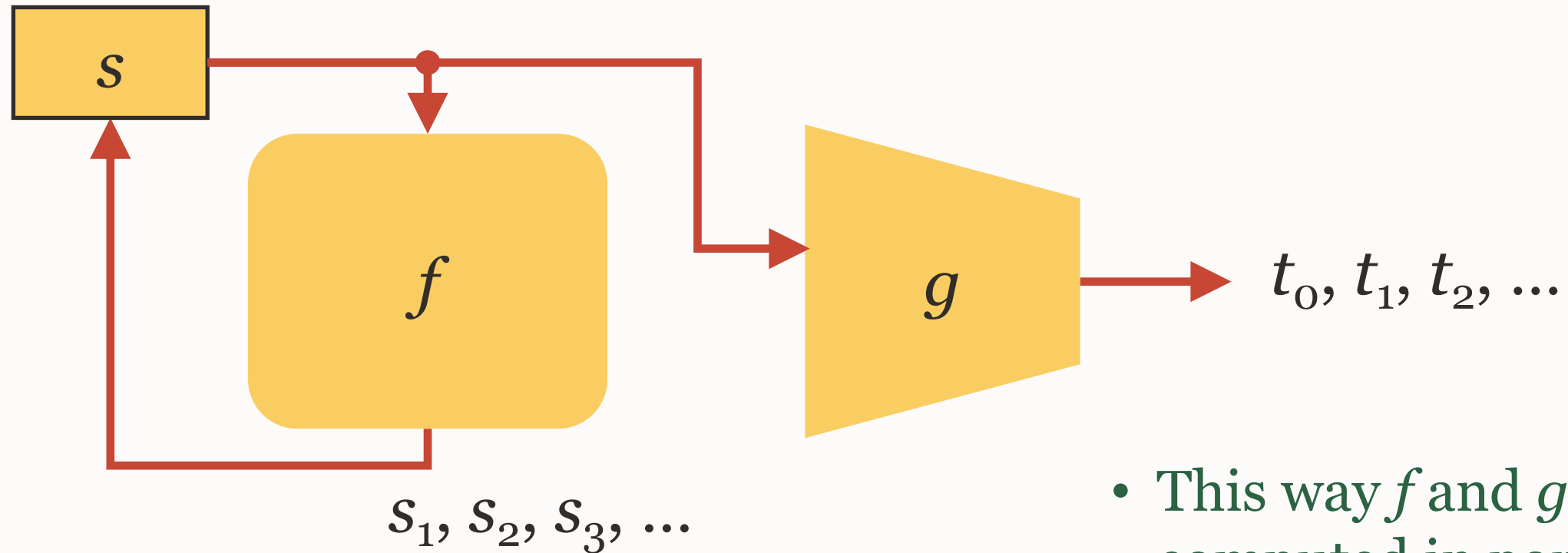


## Slight Adjustment (Engineering Hack)

initial state  $s_0$

$$s_k = f(s_{k-1})$$

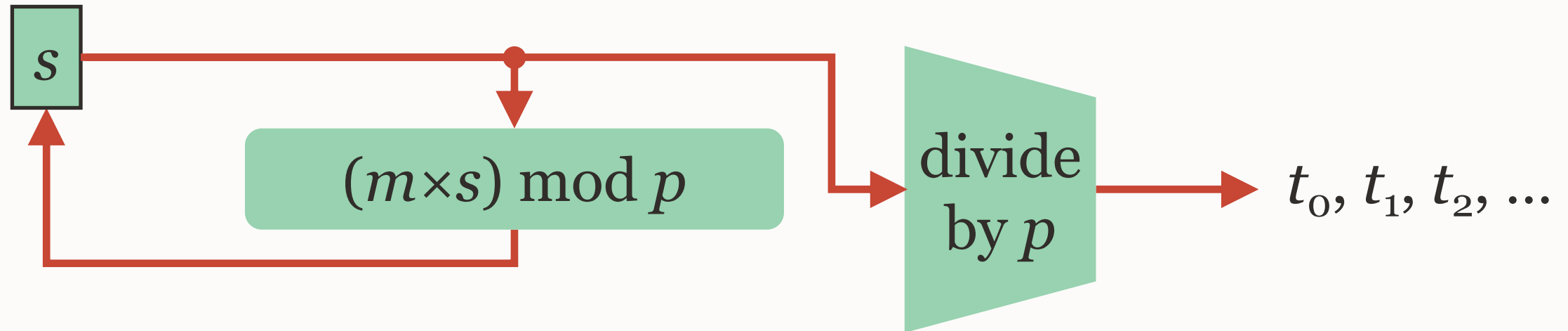
$$t_k = g(s_k)$$



- This way  $f$  and  $g$  can be computed in parallel.

## Linear Congruential PRNG with Prime Modulus

Given integer state  $s$ , modulus  $p$ , and multiplier  $m$ :

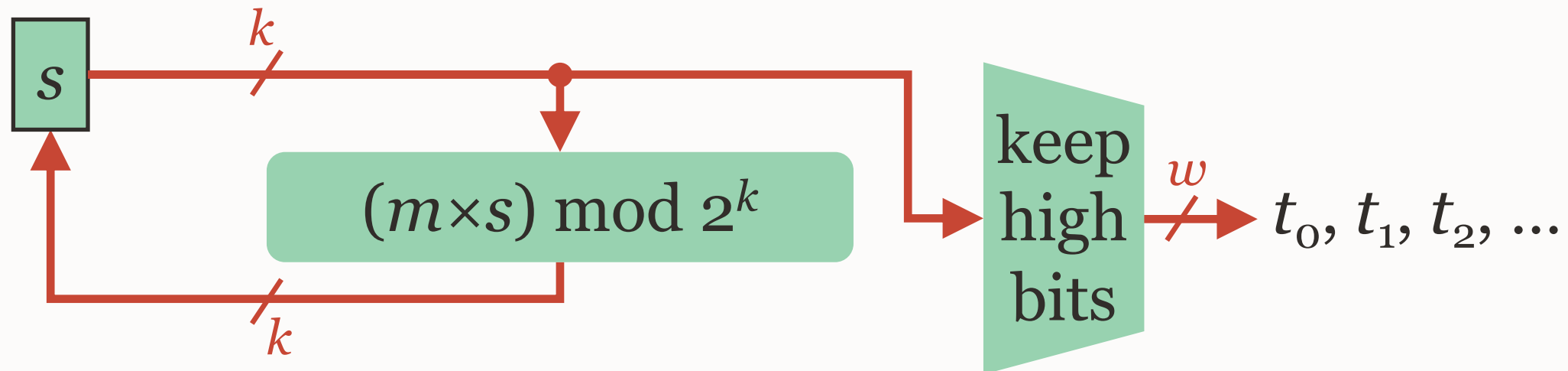


- Must choose  $s_0$ ,  $p$ , and  $m$  carefully.
- Typically  $p$  is prime.
- That's a *floating-point* divide, producing a value in  $[0.0, 1.0)$ .
- Division is expensive.



## Linear Congruential PRNG with Power-of-Two Modulus

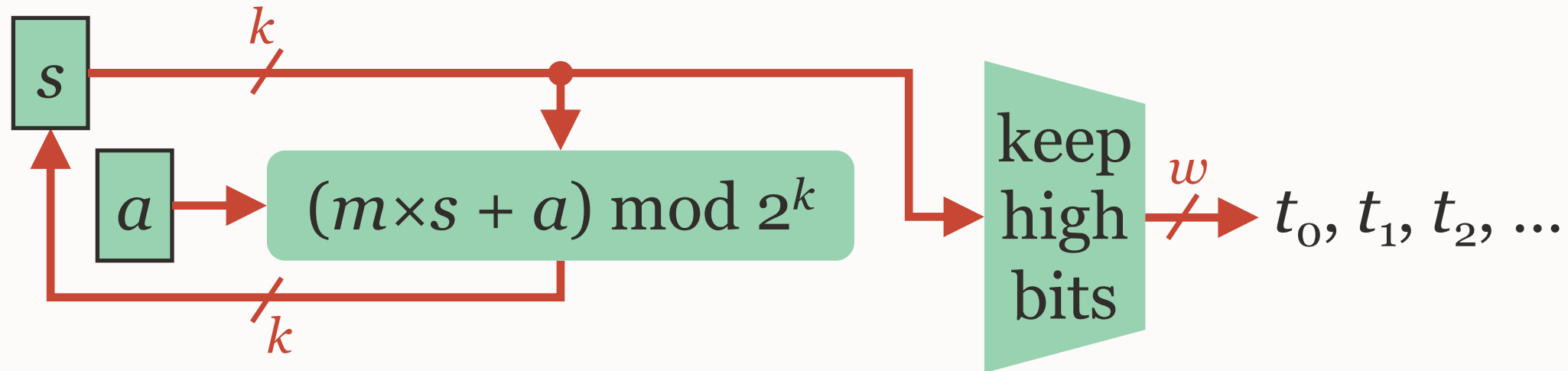
Use  $k$  bits of state, modulus  $2^k$ , and odd multiplier  $m$ :



- “keep high bits” is fast.
- Converting a bit string to floating-point is fast.
- Pretty good when  $k \geq 2w$ .
- When  $k = w$ , low bits have small period.
- Overall period cannot be larger than  $2^{k-2}$ .

## Full-Period Linear Congruential PRNG with Power-of-Two Modulus

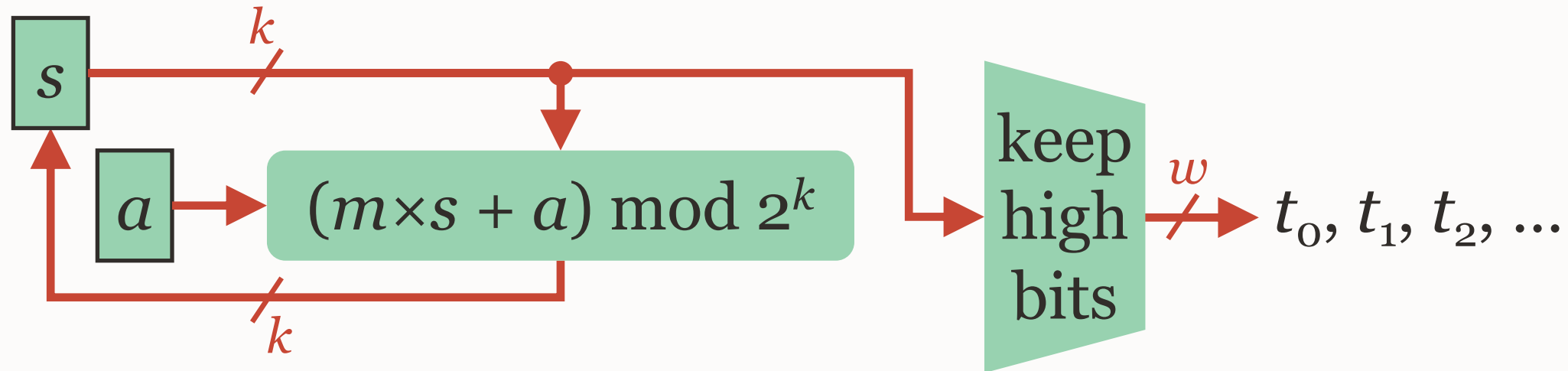
Introduce odd additive parameter  $a$ ; require  $(m \bmod 8) = 5$ :



- Now period is  $2^k$ .
- Generates all  $2^w$  values, even when  $k = w$ .
- *Exactly equidistributed.*
- But low-order bits still have low period.

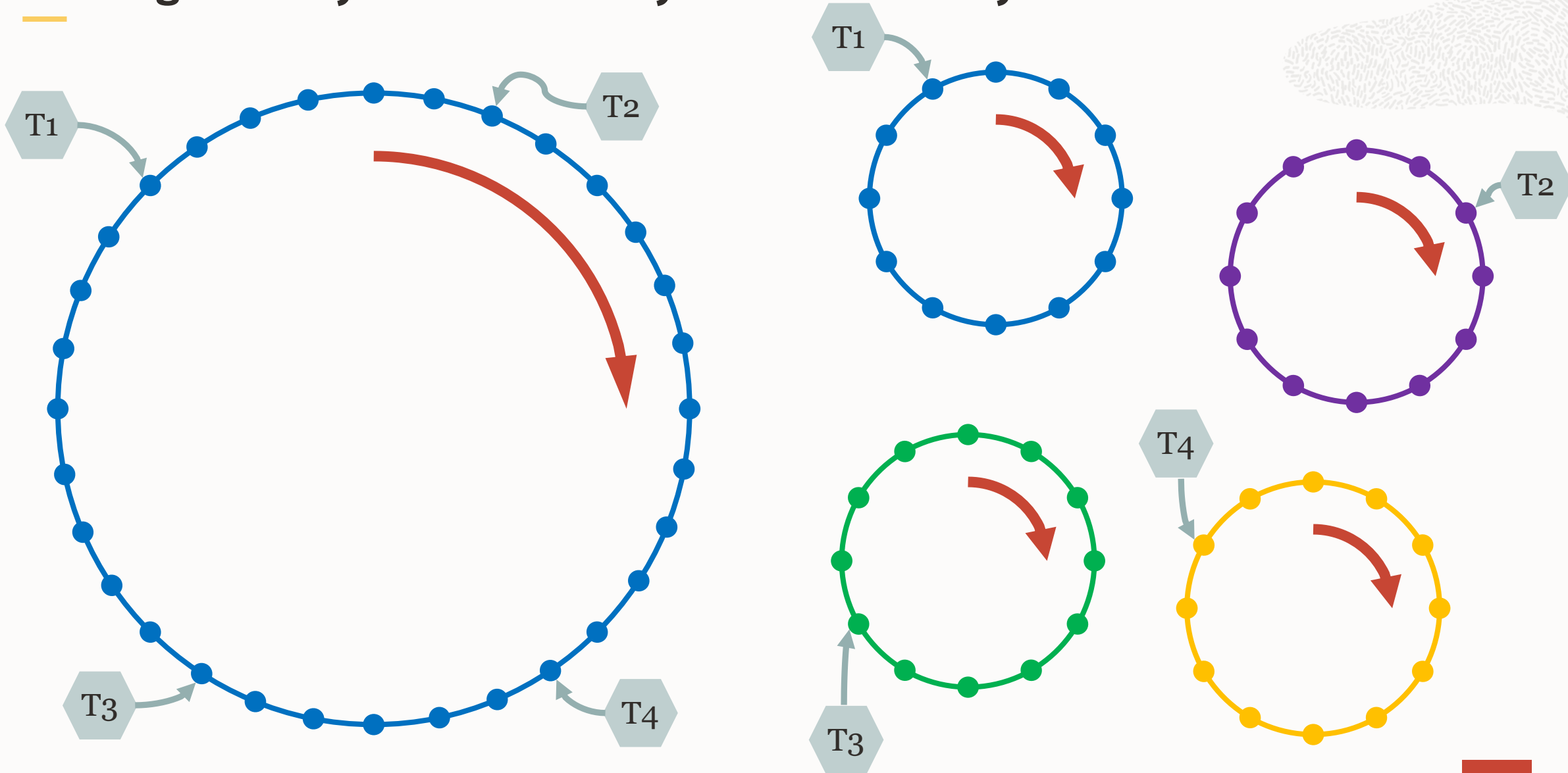
## Parallel Streams?

Idea: use  $a$  to provide independent parallel streams:



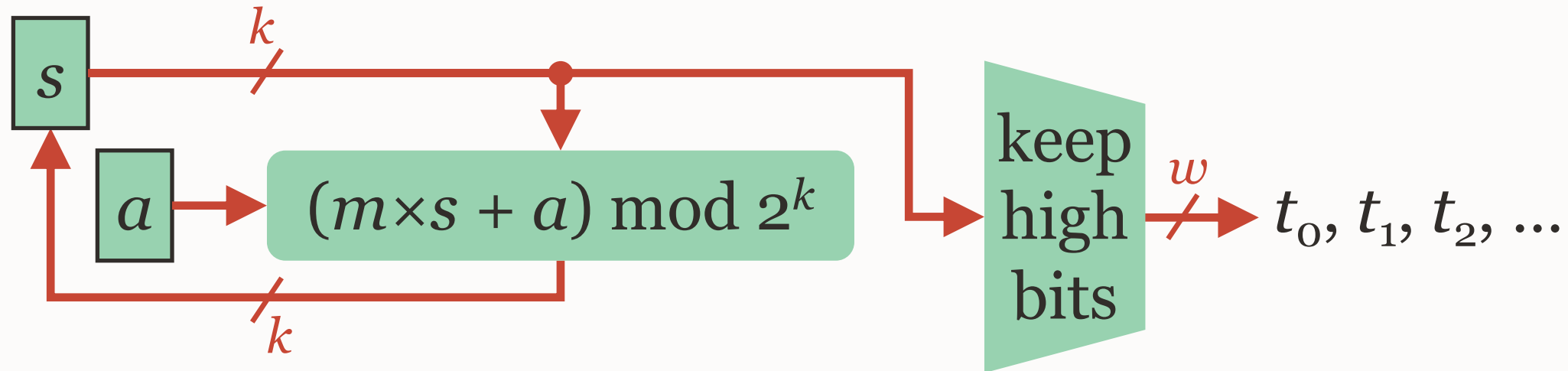
- Create many instances, each with a different value for  $a$ .
- Each parallel thread uses its own instance.

# One Big State Cycle versus Many Smaller State Cycles



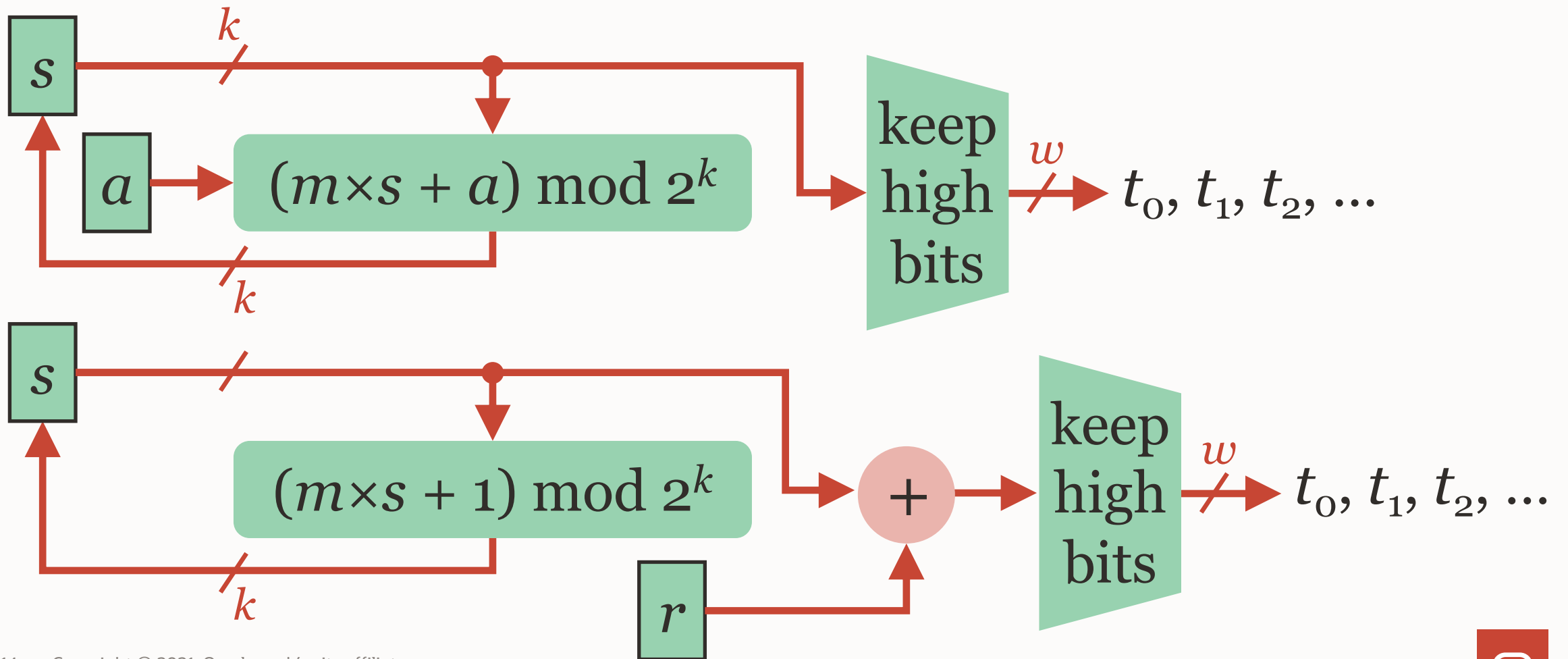
## Parallel Streams?

**FAIL!**



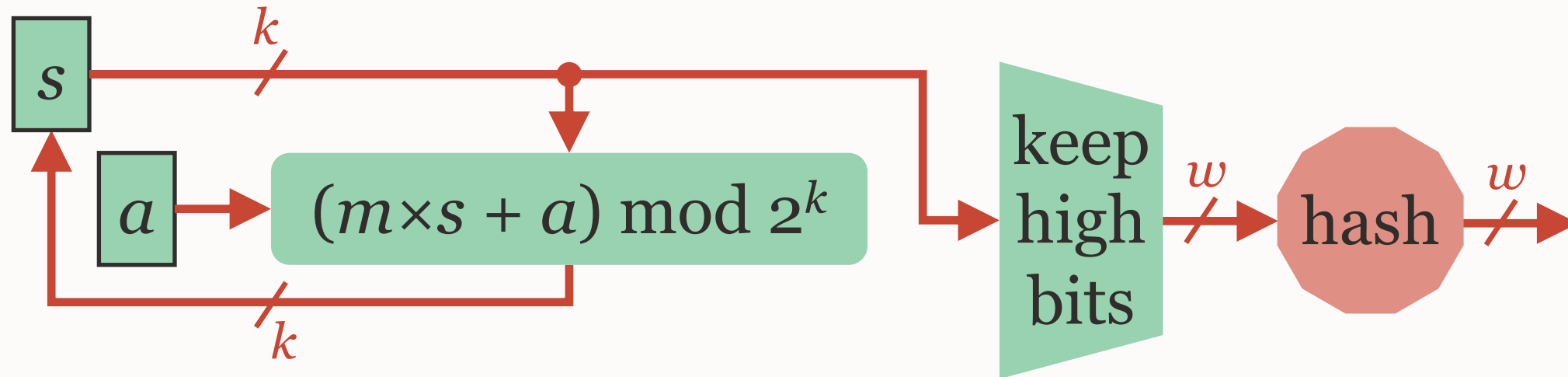
- Theorem: for any  $a$  and  $a'$  there exist constants  $i$  and  $r$  such that for all  $j$ ,  $t'_j = (t_{j+i} + r) \bmod 2^k$ .
- In visual terms: changing  $a$  doesn't change the shape of the graph that plots of  $t_j$  versus  $j$ ; it only translates it.

# Different Structures Produce Equivalent Streams



## Improving the Output Function

Add a nonlinear hashing function:

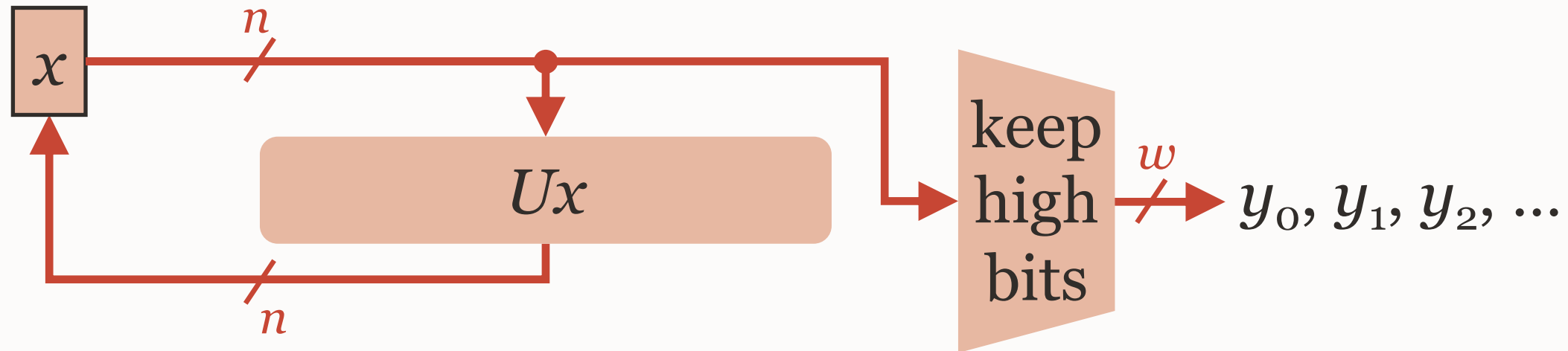


- The hash function (“bit mixer”) should have good *avalanche statistics*.
- Works great for parallel threads!
- Solves problem of low-period bits.
- But period is still only  $2^k$ .



## Getting a Large Period without Quadratic Cost

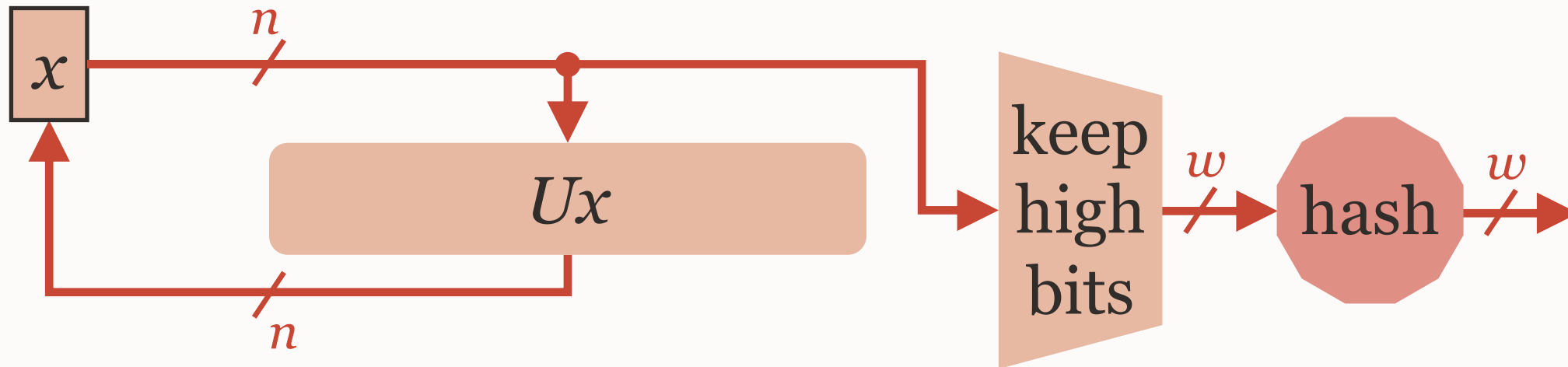
Alternate approach:  $\mathbf{F}_2$ -linear (“XOR-based”) generators.



- State vector is  $n$  **bits**; multiply by fixed  $n \times n$  **bit matrix**  $U$ .
- Overall period can be as large as  $2^n - 1$  ( $x$  is never all-zeroes).
- Output is  $(n/w)$ -equidistributed.
- Choosing  $U$  carefully allows **constant-time execution** using  $w$ -bit SHIFT/ROTATE/XOR instructions.

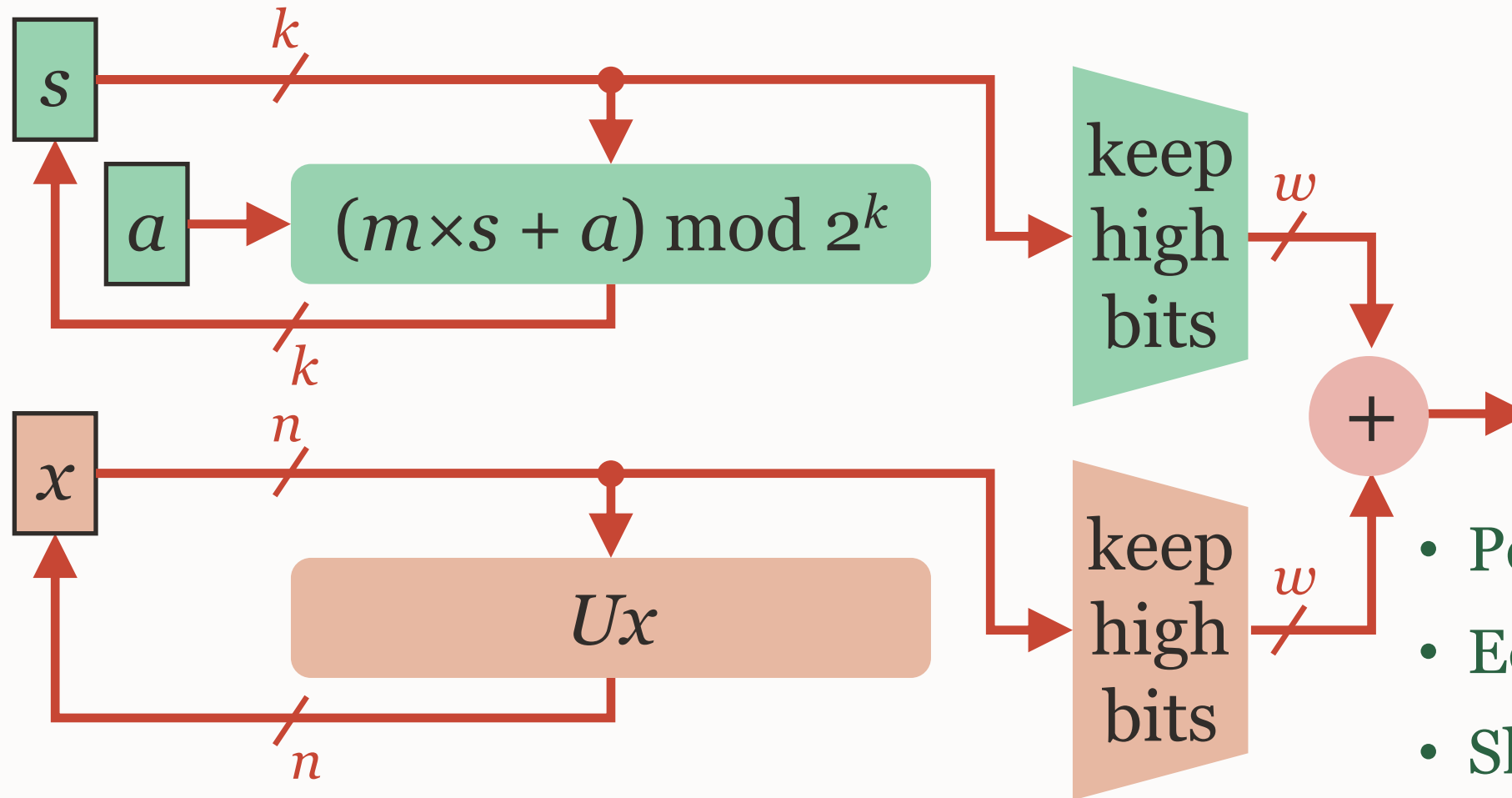
## Once Again, a Hash Function Helps

Alternate approach:  $\mathbf{F}_2$ -linear generators.



- These also have weaknesses, but adding a hash function works.

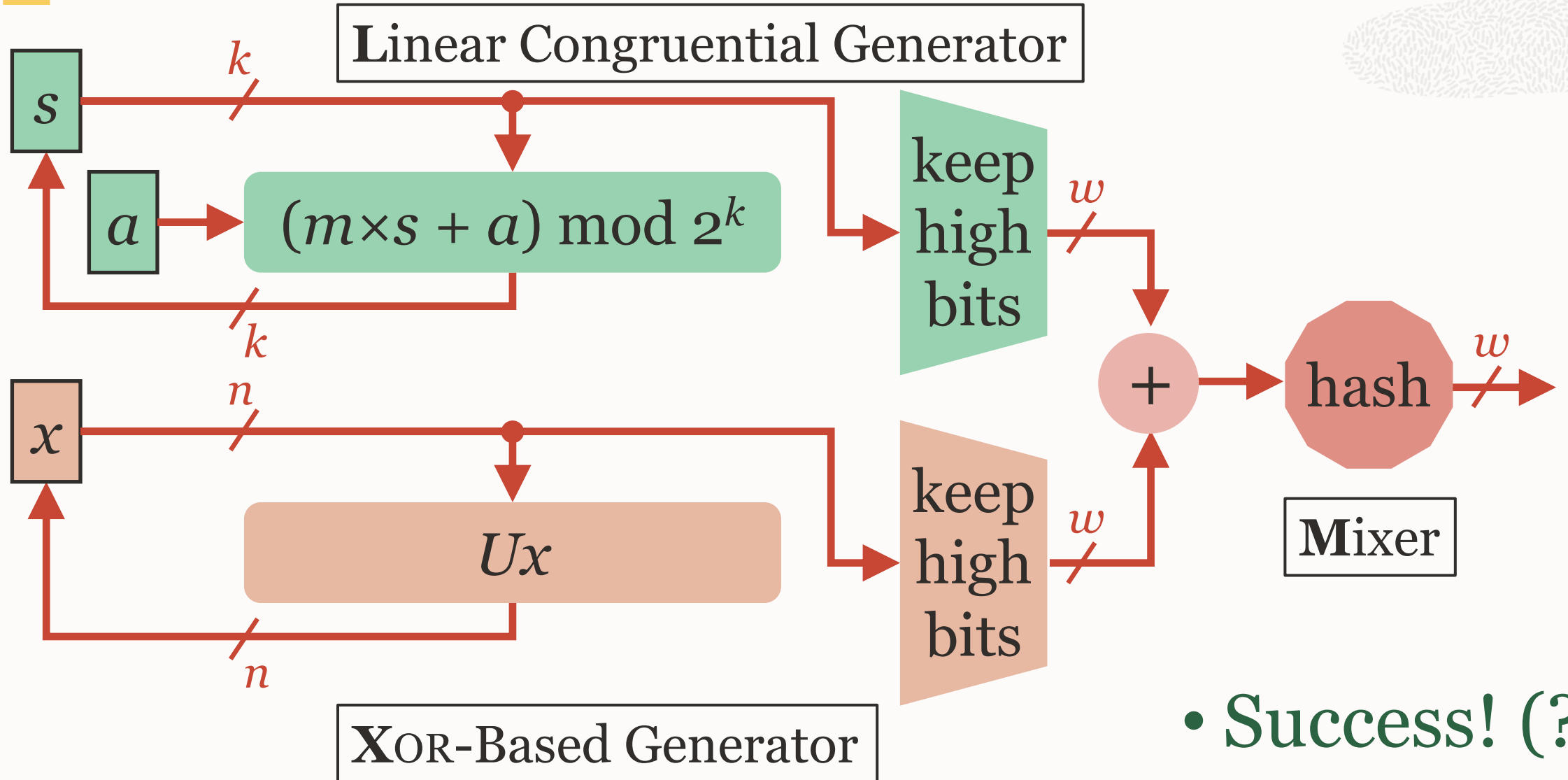
# Compound PRNG Algorithm



- Period is  $2^k(2^n - 1)$ .
- Equidistributed.
- Slightly weak.

• “if the numbers are not random, they are at least higgledy piggledy”

# The LXM Algorithm (Ta-da!)



• Success! (?)

# Deployment

---

- Specific Java implementations of this algorithm are now in JDK17
- Part of a larger API design that includes a new RandomGenerator interface
  - Easier for new PRNG algorithms to be created
  - Easier for applications to switch among PRNG algorithms
  - Also includes versions of the XOR-based xoroshiro and xoshiro algorithms

## Wait—What Makes LXM “Splittable”?

- The idea is that one can use an instance of the algorithm to create (the state for) a new one that is *statistically independent* (this is a generalization of the well-known idea of jumping to a randomly chosen point within in a single state cycle)
- This idea was appeared in the SPLITMIX algorithm (OOPSLA 2014)
  - Just generate new state data “at random”—but this was derived by stepwise refinement of the DOTMIX algorithm (PPoPP 2012)
  - Then reject certain state configurations known to be *weak*
  - Deployed as Java class SplittableRandom in JDK8
  - Unfortunately, other configurations also turned out to be weak
- LXM also splits by creating new instances “at random”
  - But we have good theoretical and empirical reasons to believe that there are no weak configurations (**we could be wrong**)

## Other Strengths of the LXM Algorithm

- Thanks to the **L**inear **C**ongruential Generator:
  - The  $w$ -bit results are exactly equidistributed (period is a multiple of  $2^w$ )
  - Additive parameter  $a$  makes it easy to provide independent parallel streams
  - Greatly improves the tuple equidistribution of the XBG
- Thanks to the **X**OR-Based Generator:
  - Period can be made very large without a large speed penalty
  - If XBG  $(n/w)$ -tuples are equidistributed, so are the LXM  $(n/w)$ -tuples
- Thanks to the **M**ixing function:
  - Eliminates linear artifacts, especially low-period low-order bits
  - Crucial to independent parallel streams: in effect,  $a$  selects the hash function

A simple, even incremental idea, but apparently not in the prior literature.  
We've seen many combinations of two of these elements before, but not all three.



## Contributions of This Paper

- Explaining why these specific components were chosen and why they should be combined in a specific way
- Analyzing certain properties of the combination
  - Period
  - Equidistribution
  - Probability of “accidental” correlations
- Comparing this algorithmic structure to prior work
  - See PRNG history in §1 and Related Work in §11 of the paper.
- Extensive quality testing (using TestU01 and PractRand test suites)
- Studies of scaling
  - Testing up to  $2^{24}$  parallel streams, using various splitting strategies
  - Testing very small versions of the algorithm (48 bits of state)
- Timing tests (LXM is indeed “almost as fast” as SPLITMIX)