FPGA-Optimised Uniform Random Number Generators using LUTs and Shift Registers

David B. Thomas and Wayne Luk
Imperial College London
{dt10,w.luk}@imperial.ac.uk

Achievements

- New FPGA-specific Random Number Generator (RNG)
  - Provide: hardware architecture + instantiation algorithm
- Architecture: use bit-wide LUTs and Shift Registers
  - Allow large period and good mixing (statistical quality)
  - Provide high clock rates with low resource utilisation
- Instantiation Algorithm: based on linear recurrences
  - Describe RNG instances using 5 parameters
  - Paper includes code generator and bit-exact simulator
- Complete RTL description of RNG instances
  - Open source VHDL available online

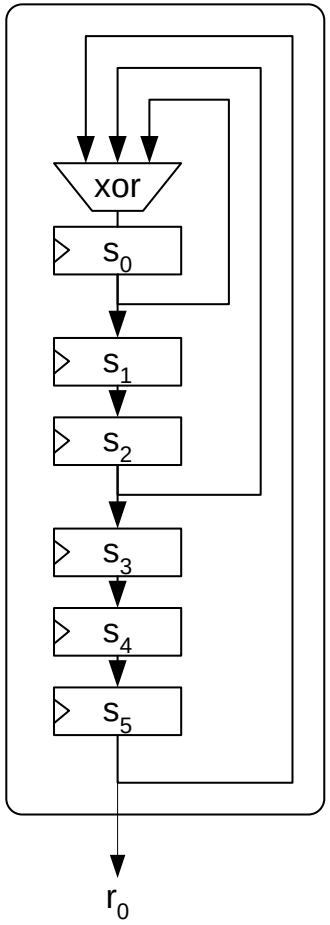
Motivation

- Monte-Carlo simulation works very well on FPGAs
  - Fine-grain parallelism: pipeline within simulator
  - Coarse-grain parallelism: instantiate parallel simulators
- Monte-Carlo is important, and becoming more so
  - Complex models often have no realistic analytical solution
  - Stochastic models can be much easier to describe
- But: Monte-Carlo is very sensitive to RNG quality
  - Obscure statistical biases can wreck simulation results
  - Parallel software guys are acutely aware of this
  - FPGA application designers need to take this into account
- Other applications also require good quality RNGs

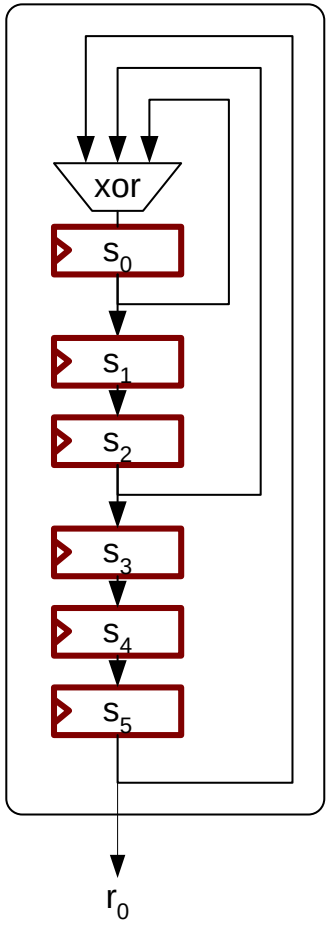
Linear Feedback Shift Registers

- The “classic” hardware uniform RNG
  - Like a classic car: poor efficiency; tends to break down
- Known outside hardware as a “GFSR using XOR”
  - Usually prefixed by “For the love of god, don’t use a ...”
  - Largely discredited in the scientific software world
- Basic principles useful in most FPGA RNGs
  - Lets take a look...

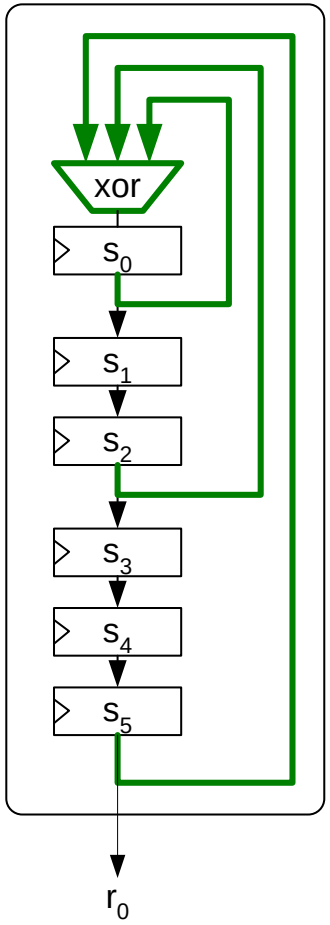
LFSR: The Architecture



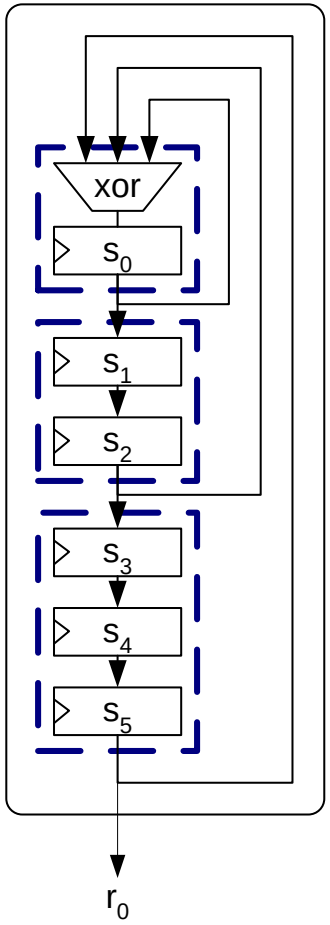
LFSR: The Architecture



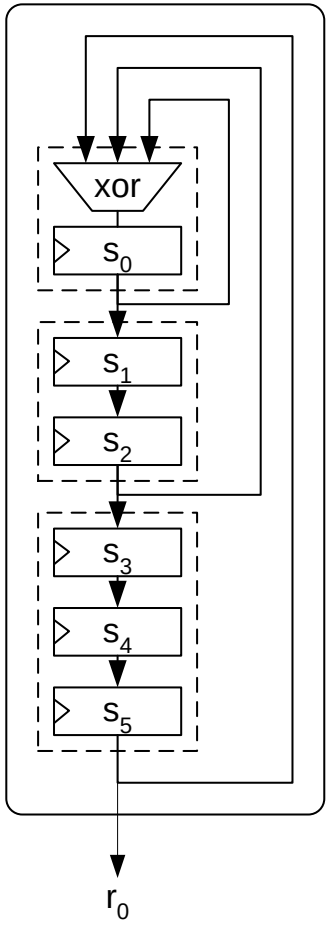
LFSR: The Architecture



LFSR: The Architecture



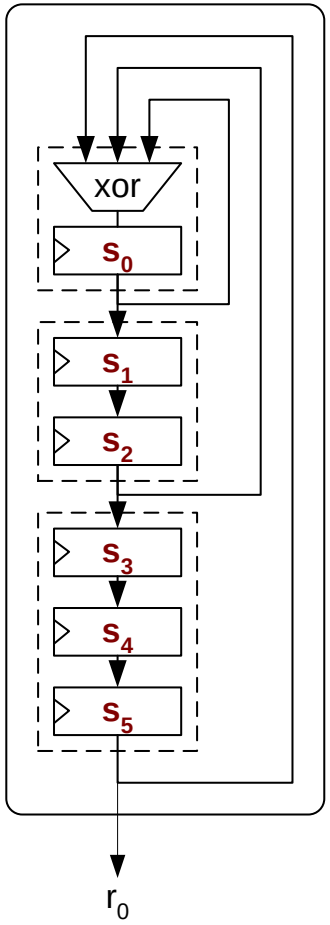
LFSR: The Theory



$$\mathbf{s'} = \mathbf{A} \times \mathbf{s}$$

$$\begin{bmatrix} s_0' \\ s_1' \\ s_2' \\ s_3' \\ s_4' \\ s_5' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

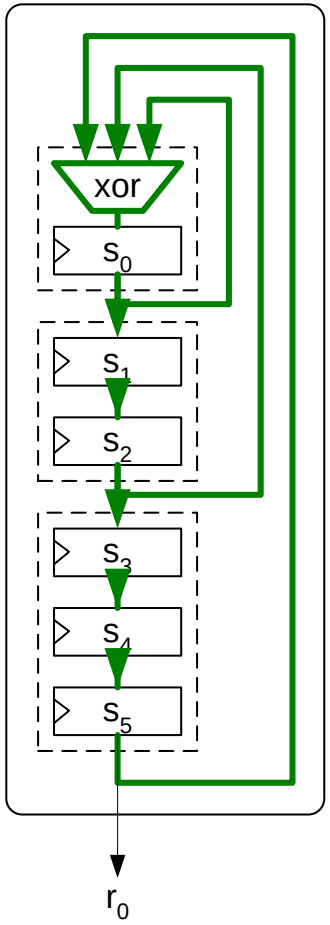
LFSR: The Theory



$$s' = A \times s$$

$$\begin{bmatrix} s_0' \\ s_1' \\ s_2' \\ s_3' \\ s_4' \\ s_5' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

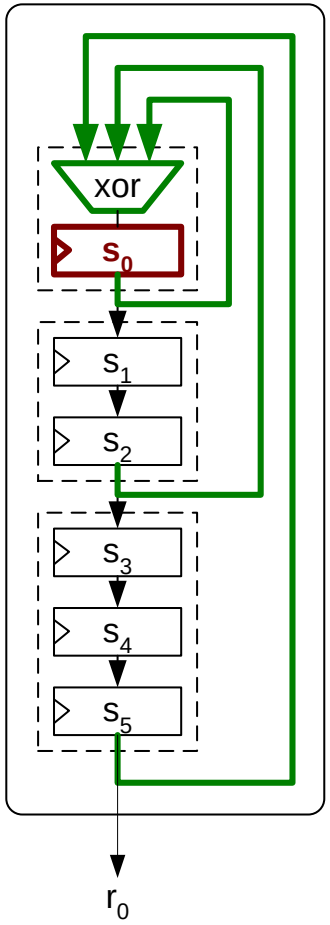
LFSR: The Theory



$$\mathbf{s'} = \mathbf{A} \times \mathbf{s}$$

$$\begin{bmatrix} s_0' \\ s_1' \\ s_2' \\ s_3' \\ s_4' \\ s_5' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

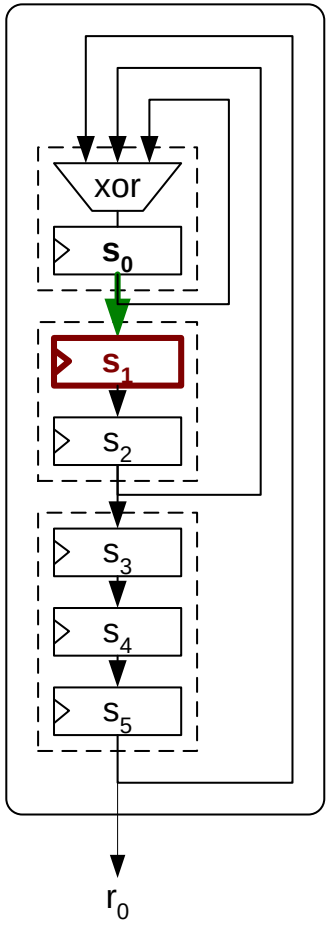
LFSR: The Theory



$$s' = A \times s$$

$$\begin{bmatrix} s_0' \\ s_1' \\ s_2' \\ s_3' \\ s_4' \\ s_5' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

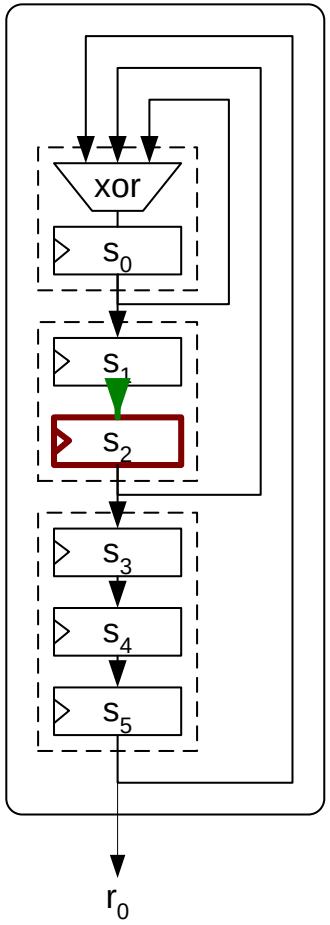
LFSR: The Theory



$$s' = A \times s$$

$$\begin{bmatrix} s_0' \\ s_1' \\ s_2' \\ s_3' \\ s_4' \\ s_5' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

LFSR: The Theory



$$\mathbf{s'} = \mathbf{A} \times \mathbf{s}$$

$$\begin{bmatrix} s_0' \\ s_1' \\ s_2' \\ s_3' \\ s_4' \\ s_5' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

General Theory: Linear Recurrences

- The matrix **A** defines a linear recurrence (mod-2)
  - Let $\mathbf{s}_0$ be the initial state of the LFSR
  - After one cycle: $\mathbf{s}_1 = \mathbf{A}\,\mathbf{s}_0$
  - After two cycles: $\mathbf{s}_2 = \mathbf{A}\,\mathbf{s}_1 = \mathbf{A}\,(\mathbf{A}\,\mathbf{s}_0) = \mathbf{A}^2\,\mathbf{s}_0$
  - After *i* cycles: $\mathbf{s}_i = \mathbf{A}^i\,\mathbf{s}_0$
- Eventually the sequence must repeat (finite state)
  - For an *n* bit state the maximum possible period is 2^n-1
  - Can determine period by analysing **A**
- Challenge: given requirements
  - Maximise period
  - Find best statistical quality subject to maximal period

LFSRs: Why are they so bad?

- Quality : have known statistical quality problems
- Period : LFSRs only make sense for n < 128
  - A period of 2^{128}-1 is worryingly low for parallel simulations
  - Makes randomised seeding of parallels RNGs hazardous
- Efficiency : a 1-bit LFSR requires many resources
  - Needs four or five LUT-FF elements for each bit
- Scalability : the LFSR only provides one bit/cycle
  - Most applications need far more bits per cycle
  - Multiple bits requires independent parallel LFSRs
    - Parallel LFSRs = GFSR+XOR : ***bad*** reputation
- *Please don't use LFSRs for parallel Monte-Carlo...*

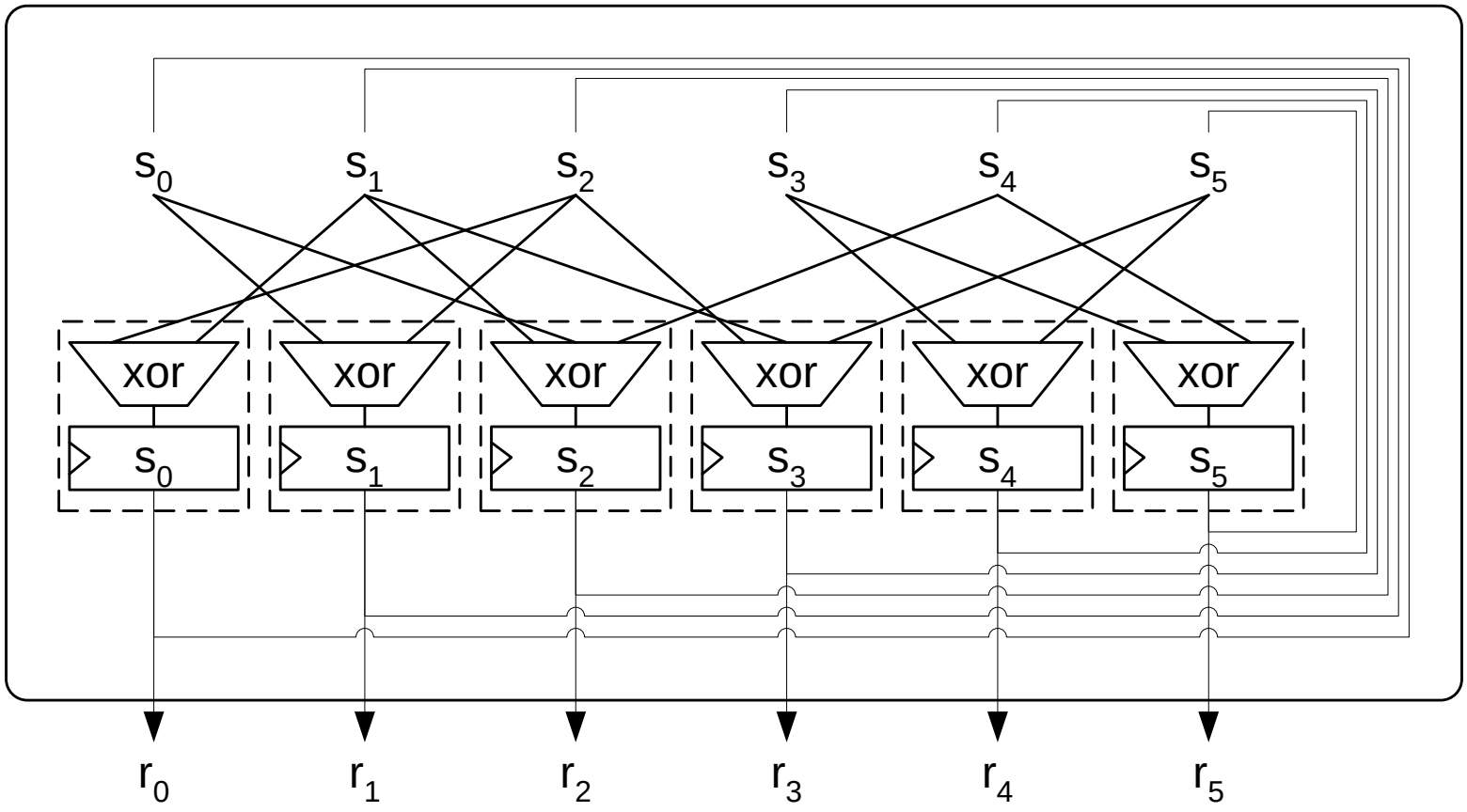
Previous FPGA RNGs: Software RNGs

- Take a software generator, implement in RTL
  - Popular choice: take advantage of existing algorithms
  - Mersenne Twister, SPRNG, lagged Fibonacci
- Software generators often not efficient in hardware
  - Use different operation costs: no bit-level ops or RAMs
  - Ignore data-dependencies; may not pipeline well
- Word-based generators provide 32 or 64-bit words
  - FPGA simulations often need 128+ random bits/cycle
  - Instantiating parallel generators is a poor solution
- Common practice: optimise SW algorithm for HW
  - But: linear recurrences have subtle properties
  - e.g. attempts to increase outputs/cycle -> unknown period

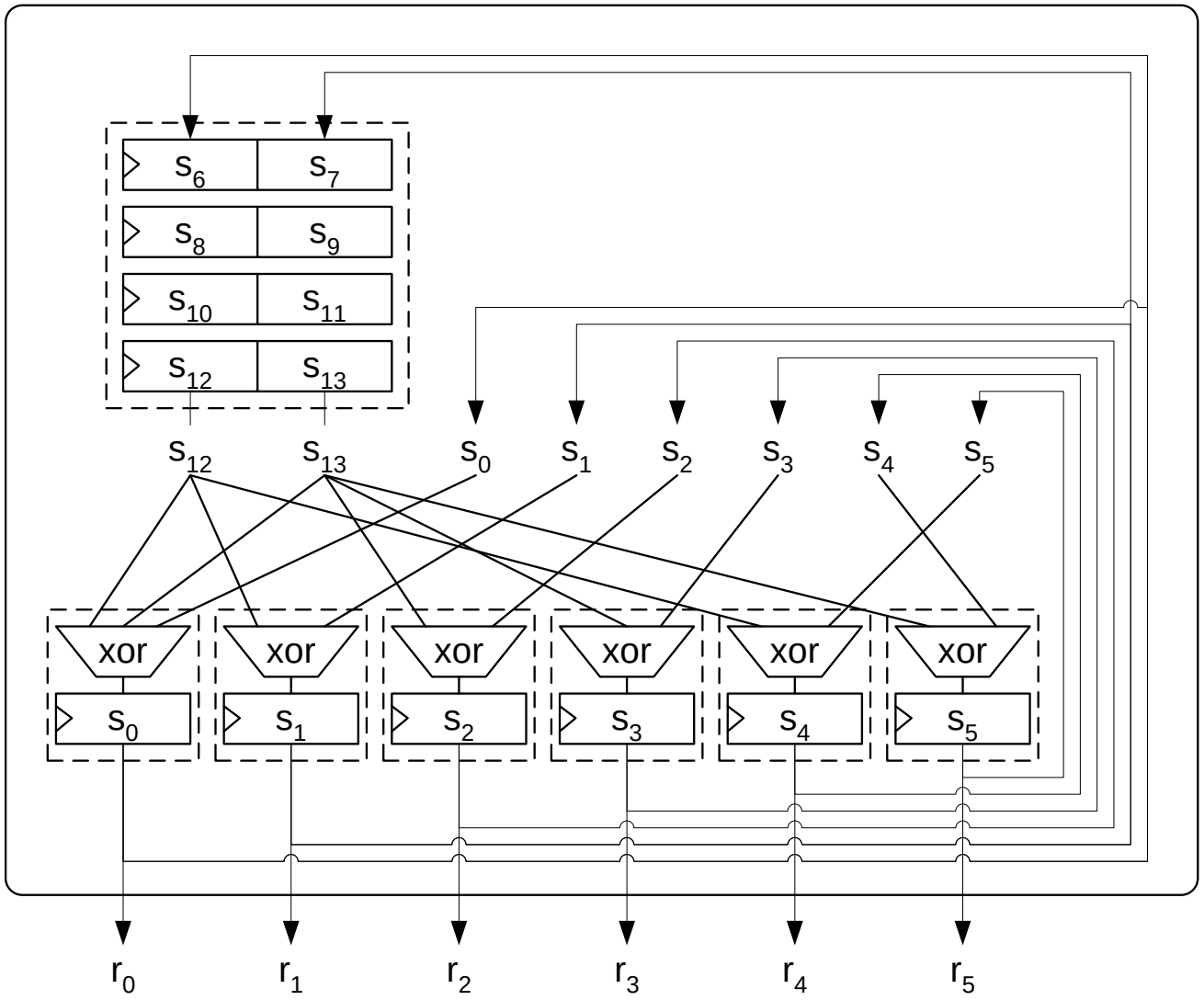
How can we create a ***new*** FPGA RNG

1. Define a general architecture for the generator
   - What sort of resources, and how are they connected?
   - What constraints are placed on the connections?
2. Create algorithm for constructing instances of RNG
   - Use random choices to make decisions about structure
3. Search for good instances of the generator
   a. Generate a huge number of candidate RNG instances
   b. Discard all candidates where constraints are not met
   c. Extract matrix A for candidate RNGs
   d. Discard RNGs which do not have maximum period
   e. Select best remaining RNG according to quality metrics

Previous FPGA RNGs: LUT-Optimised



Improve Period: LUT-FIFO



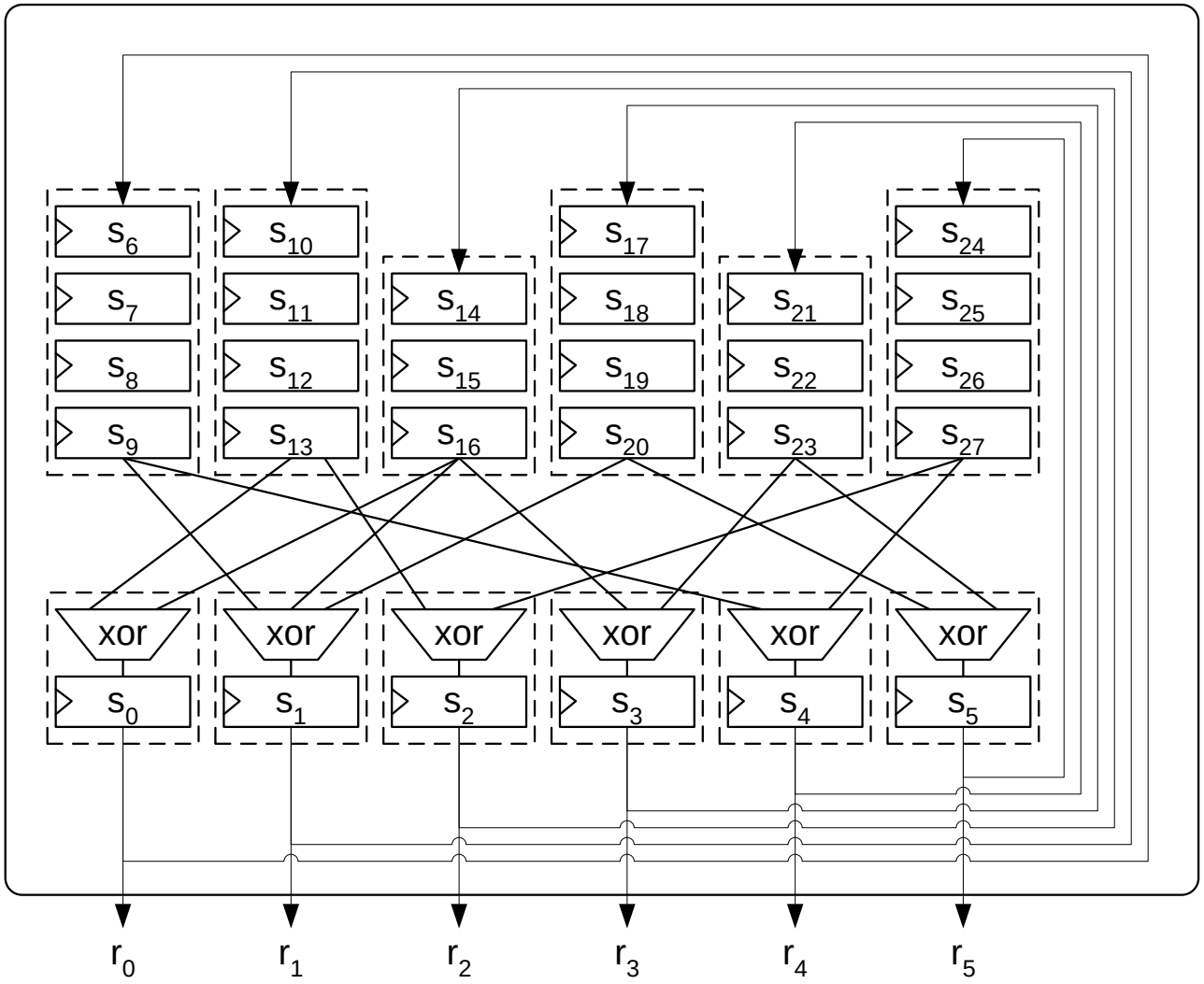
Why do we need ***another*** RNG?

- The LUT-Opt generator is very small and fast
  - Good as a building block for non-uniform generators
  - But: has some mild statistical problems
- The LUT-FIFO has superb quality and a huge period
  - Same class as Mersenne Twister, but more efficient
  - But: overkill for most applications, and requires block RAM
- Both generators are hard to distribute to users
  - Require carefully selected parameters
  - Impossible to include the code in papers
  - Full set of generators requires megabytes of VHDL

LUT-SR Generator: Goals

- Architecture - Efficient, Scalable, and Foolproof
  - Efficient : constant resources per generated bit
  - Scalable : quality and period scale with output width
  - Foolproof : no quality issues; appropriate for use anywhere
- Specification - Concise, Complete, and Open Source
  - Concise : generators described using a simple algorithm
  - Complete : ***all*** parameters and algorithms included in paper
  - Open Source : VHDL libraries made available online
- No excuses - if you're still using an LFSR... why?

The LUT-SR Architecture



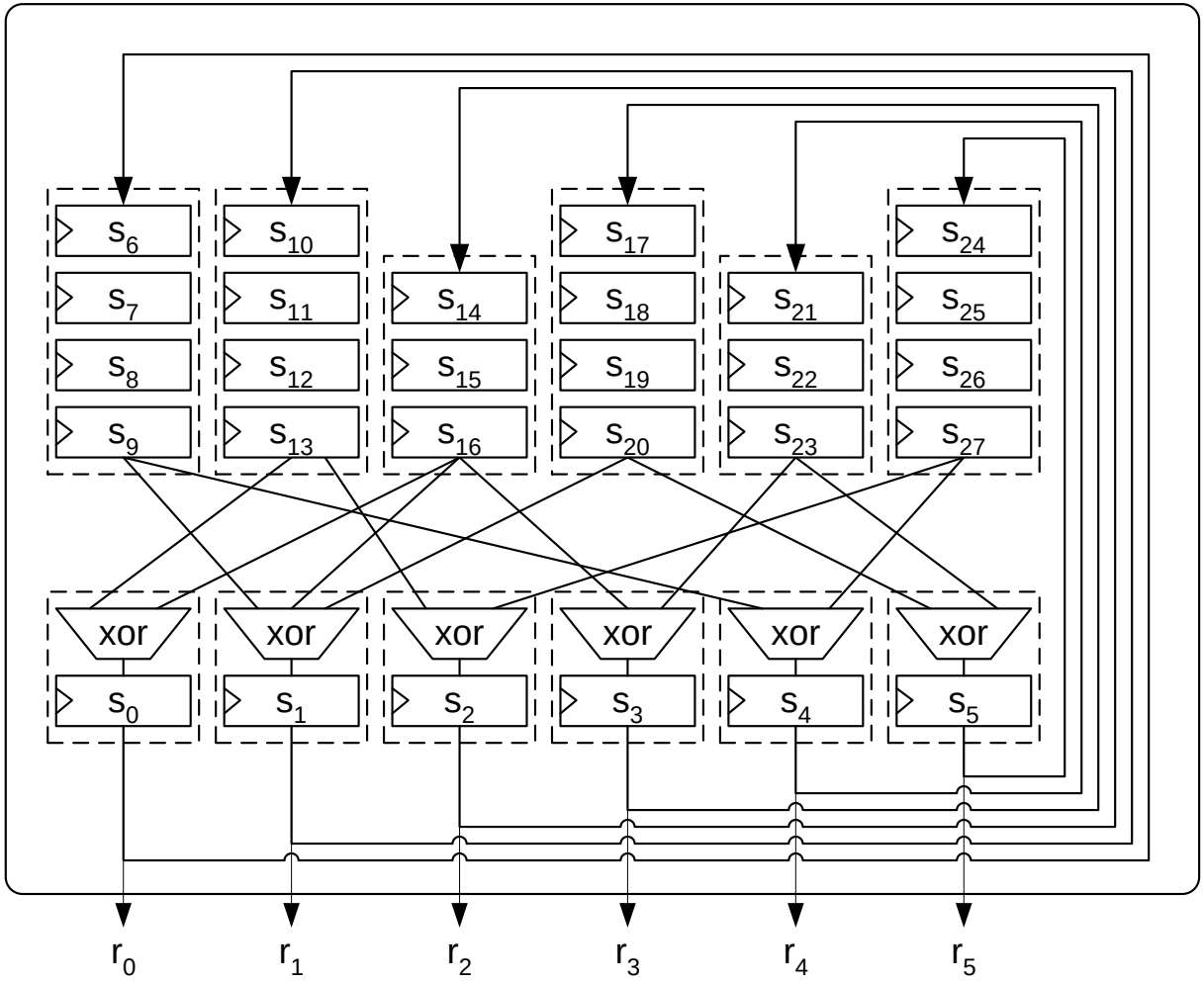
Architecture: Optimise Temporal Quality

- Outputs only depend on Shift-Register (SR) outputs
  - No direct dependency between previous and next output
  - Minimum dependency distance is depth of shortest SR
- Scale storage with output bits
  - Equidistribution - theoretical measure of auto-correlation
  - LUT-Opt: poor equidistribution due to 1:1 storage:output
  - LUT-FIFO: equidistribution decreases as outputs increase
  - LUT-SR: *equidistribution constant as output bits increase*
- Use Shift-Registers of different lengths
  - Increases mixing between bits in state
  - Improve “avalanche” property: time till one bit affects all bits

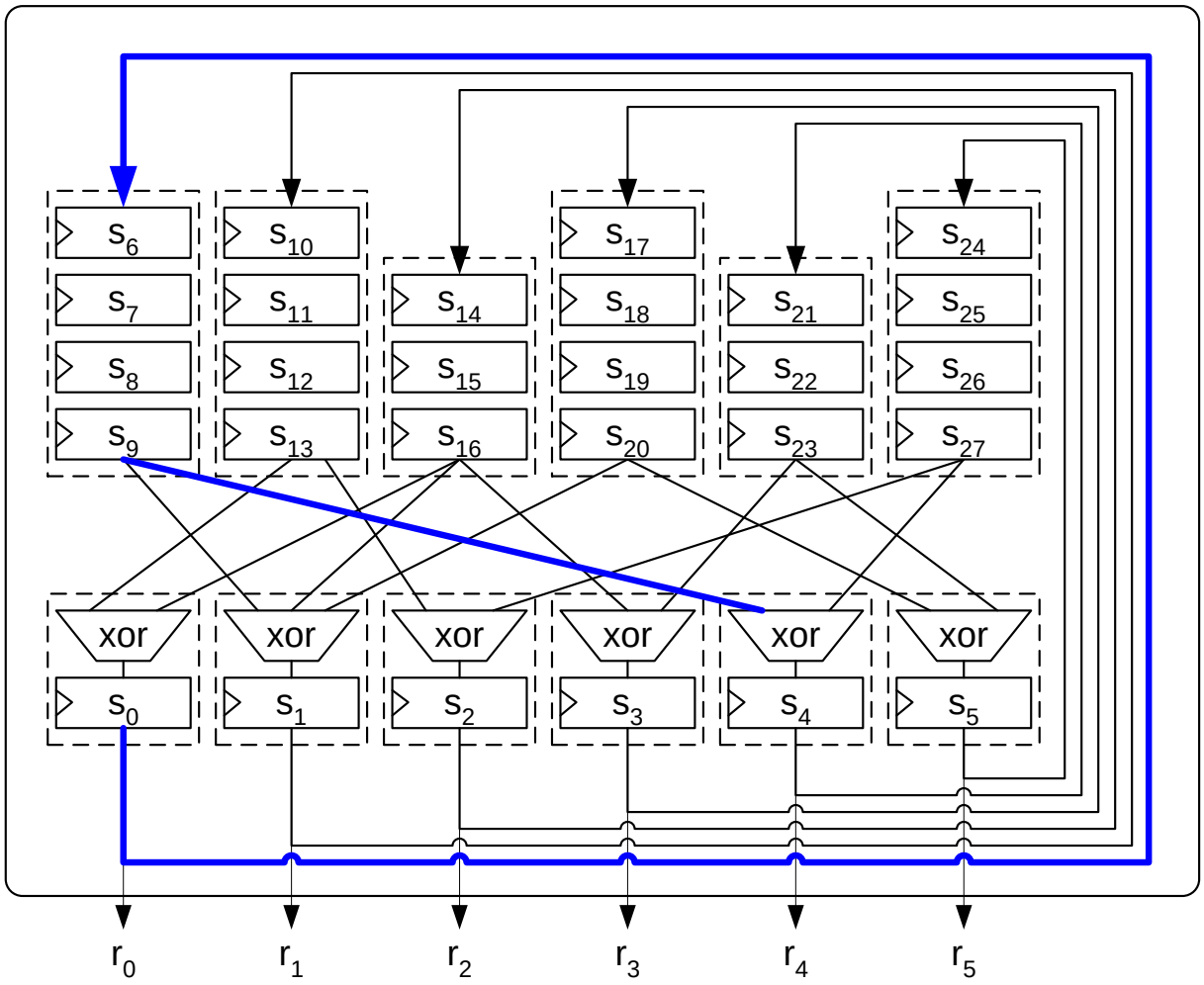
Initialisation: Managing RNG State

- Loading RNG state is important
  - We need to be able to put the RNG in a ***specific*** state
  - Parallel simulations have to manage RNG states carefully
    - Leapfrogging – single RNG sequence split among RNGs
    - Random initialisation – use random seed state for each RNG
- Loading RNG state is infrequent
  - 99.999% of the time is spent generating random bits
  - Can't spend lots of resources on initialisation
- Previous approaches tend to side-step initialisation
  - *Umm, including LUT-Opt and LUT-FIFO*
  - Suggested expensive parallel loading of state
- We can get initialisation for free, *if we are careful*

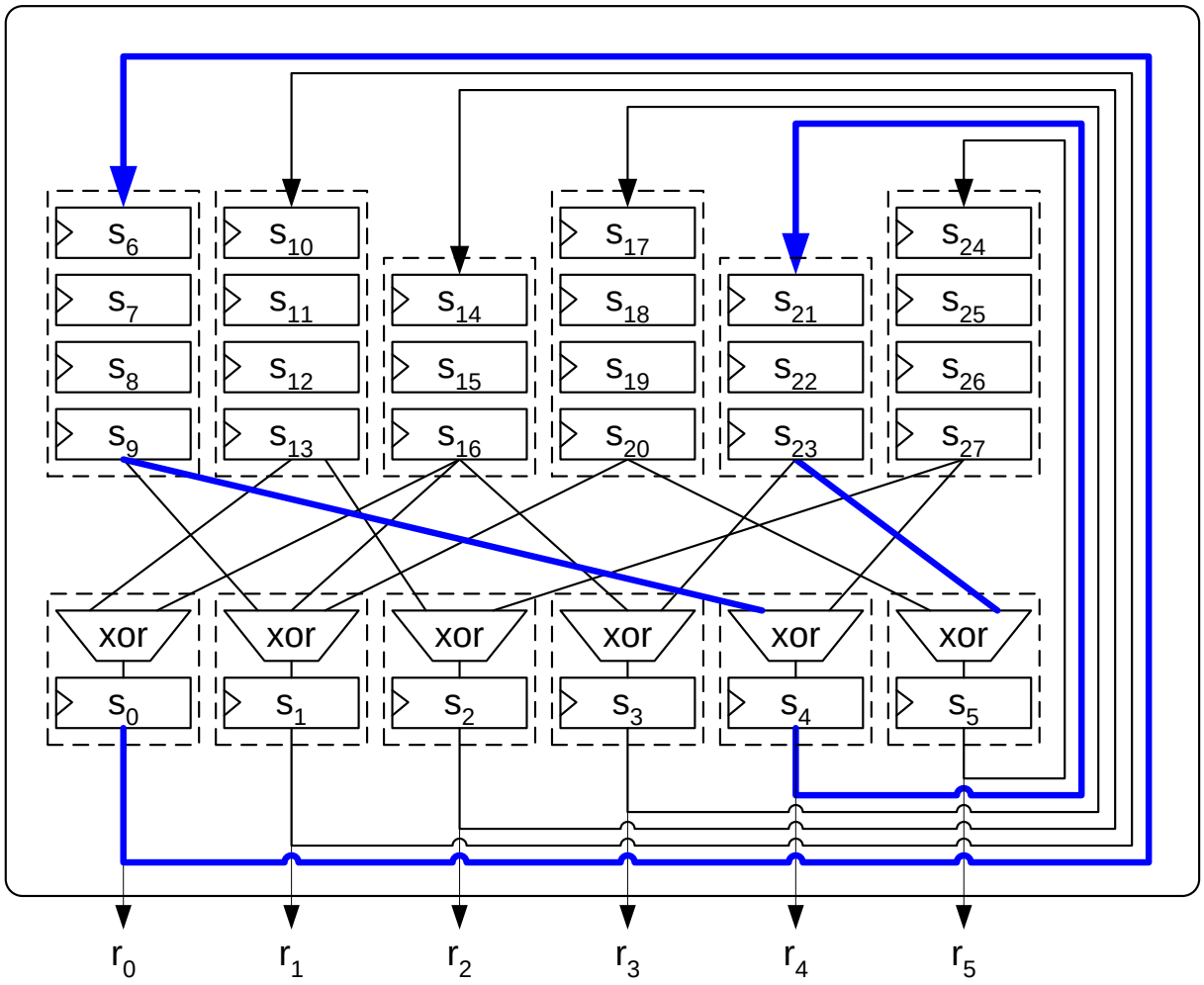
Initialisation: Finding a Cycle



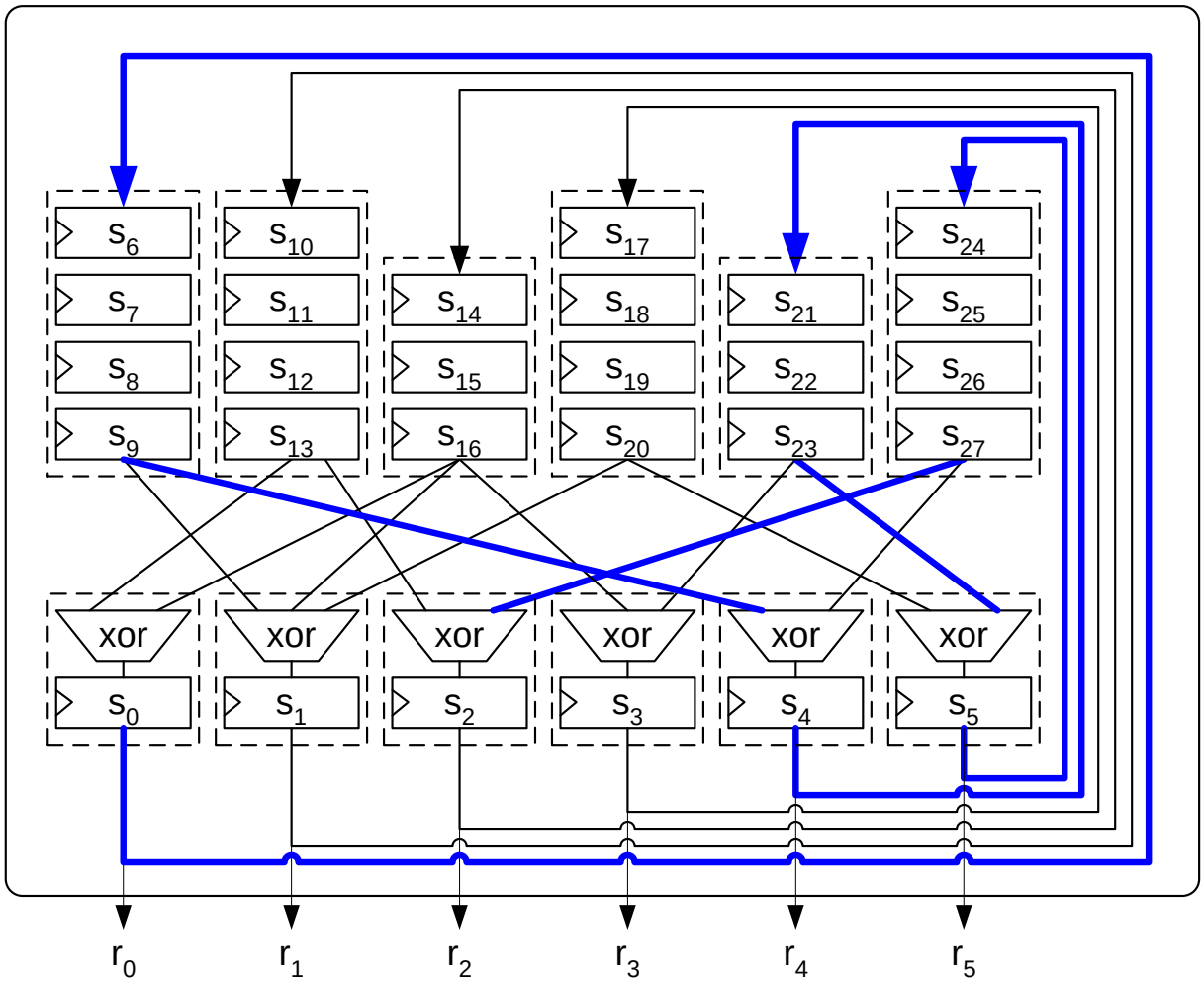
Initialisation: Finding a Cycle



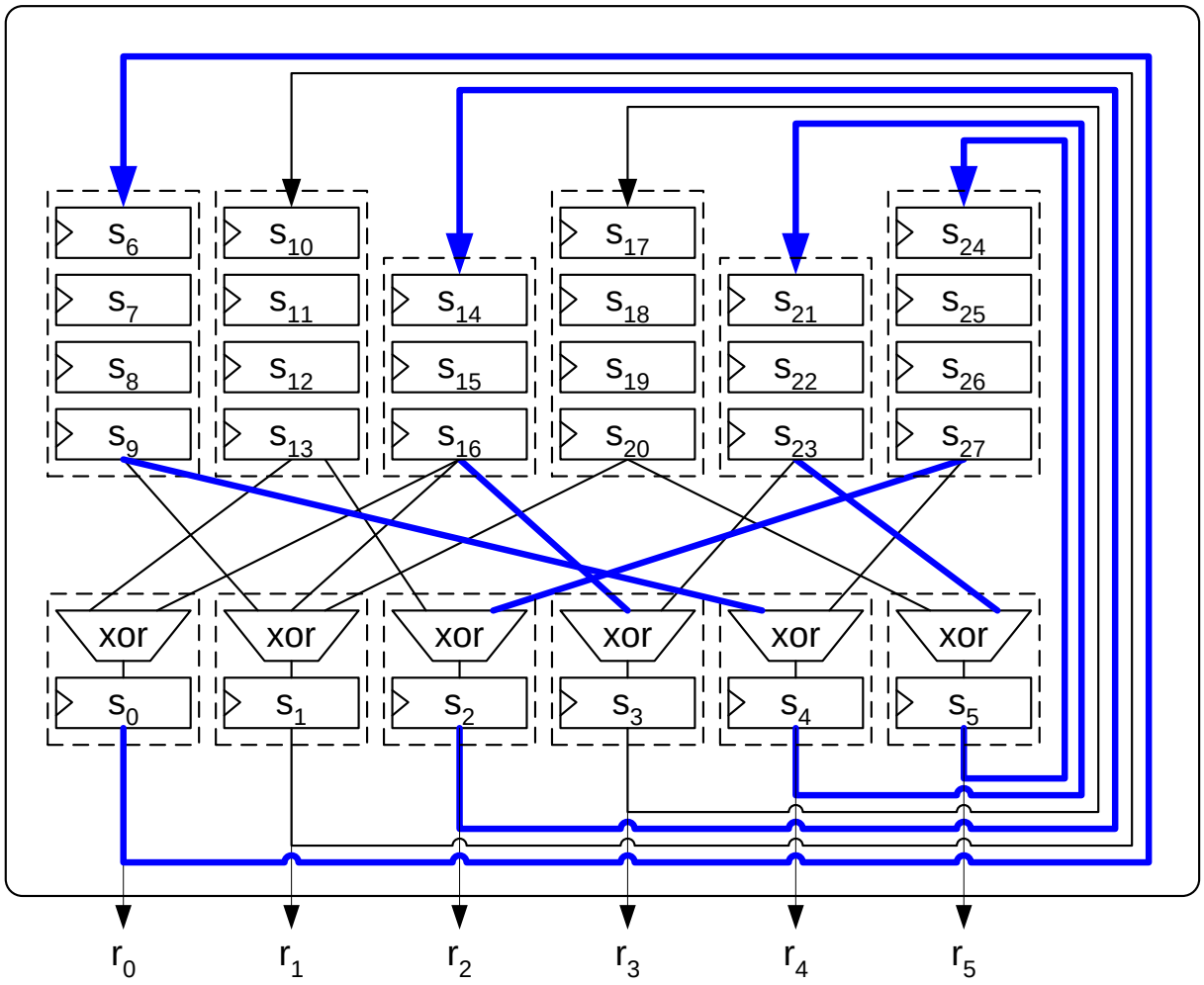
Initialisation: Finding a Cycle



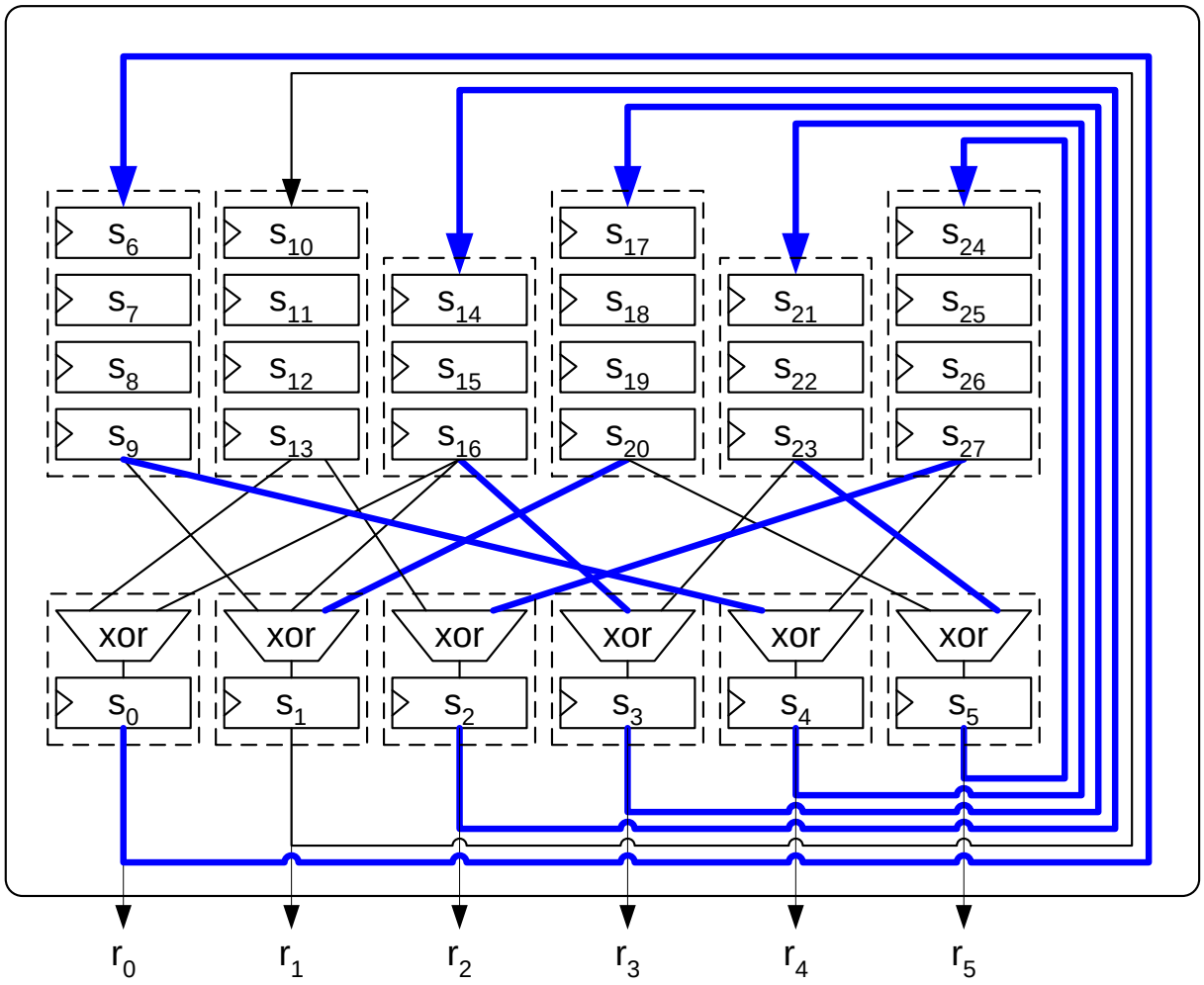
Initialisation: Finding a Cycle



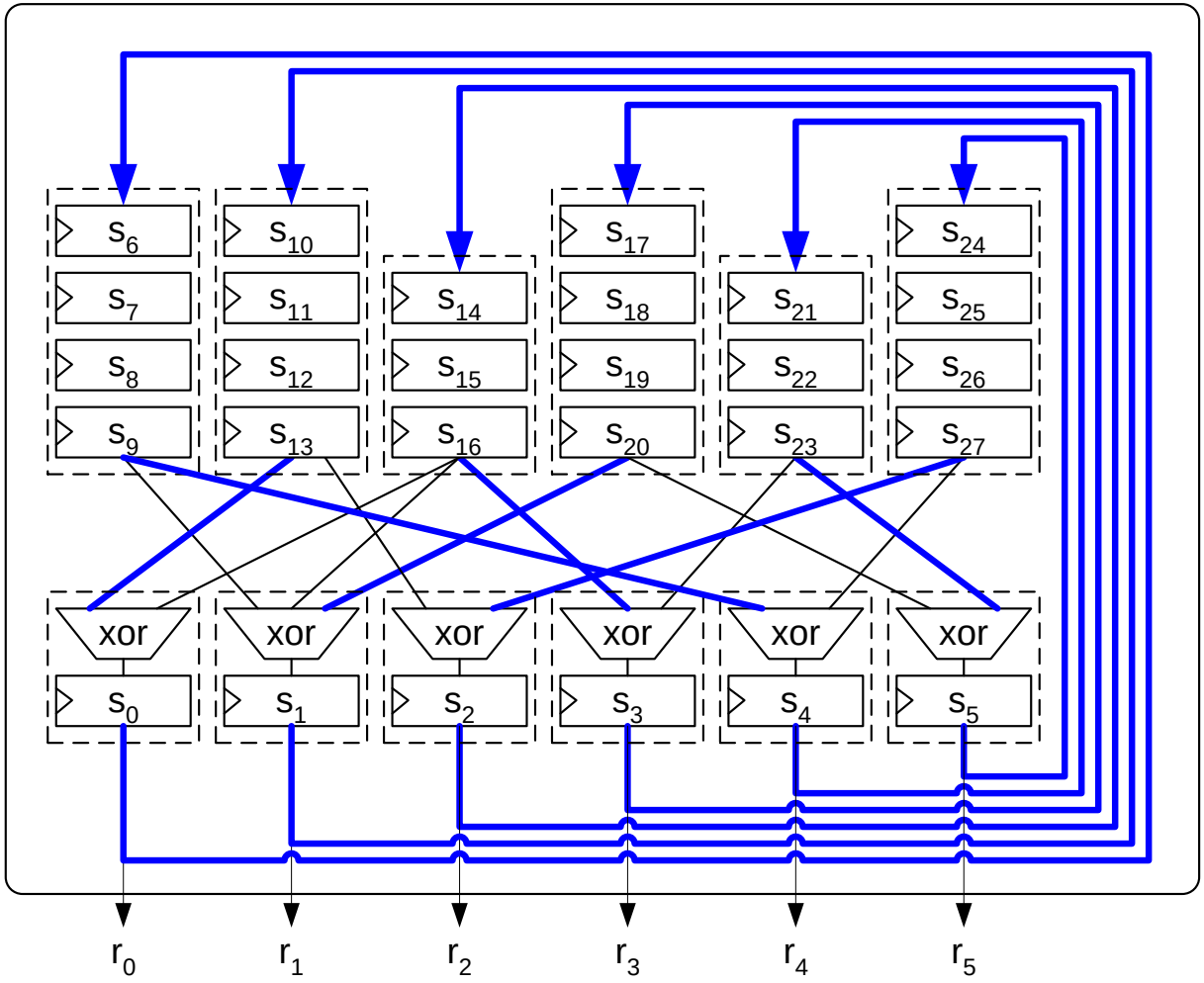
Initialisation: Finding a Cycle



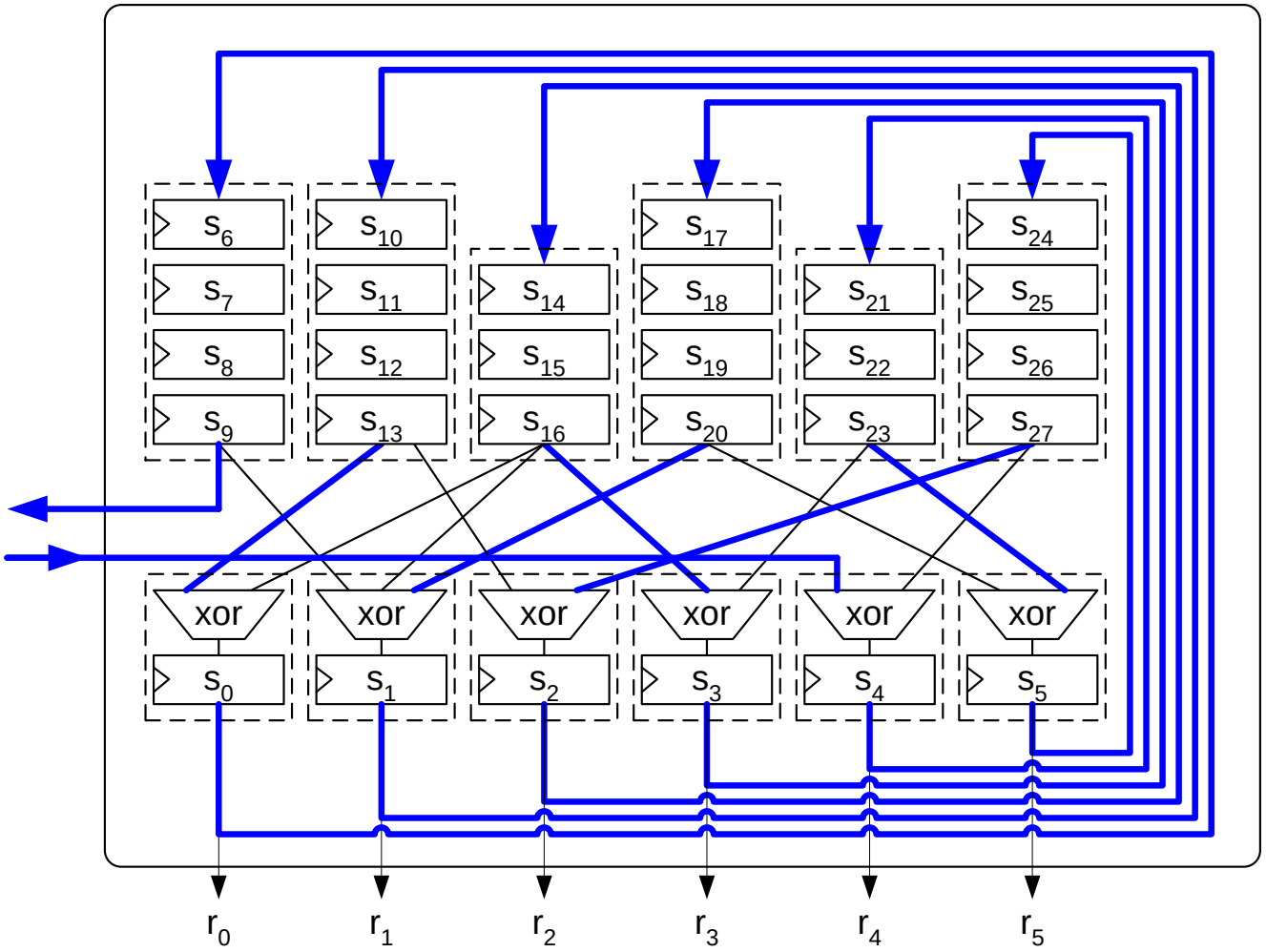
Initialisation: Finding a Cycle



Initialisation: Finding a Cycle



Initialisation: Opening Up the Cycle



How do we find the cycle?

- Previous work casually suggested “just find a cycle”
  - **cough*, my paper on LUT-Opt generators*
- Given an arbitrary generator, finding a cycle is slow
  - Well known graph problem: Hamiltonian cycle
  - Doesn’t really work for large numbers of output bits
- Have to build the cycle into basic structure of RNG
  - ***Start*** with a cycle, then add extra taps on top

Specification of LUT-SR RNGs

- Four parameters describe type of RNG
  - *n* : Number of bits in the state
  - *r* : Number of output bits per cycle
  - *t* : Number of xor inputs per bit (e.g. for 6-LUT choose *t*=5)
  - *k* : Maximum shift register length (e.g. *k*=32 for SRL32)
- Each (*n*,*r*,*t*,*k*) describes a *huge* space
  - Many possible *n* x *n* **A** matrices matching our template
- We want to pick specific generators which:
  1. Match a specific parameterisation (*n*,*r*,*t*,*k*)
  2. Have the maximum possible period of 2^n-1
  3. Have excellent statistical properties

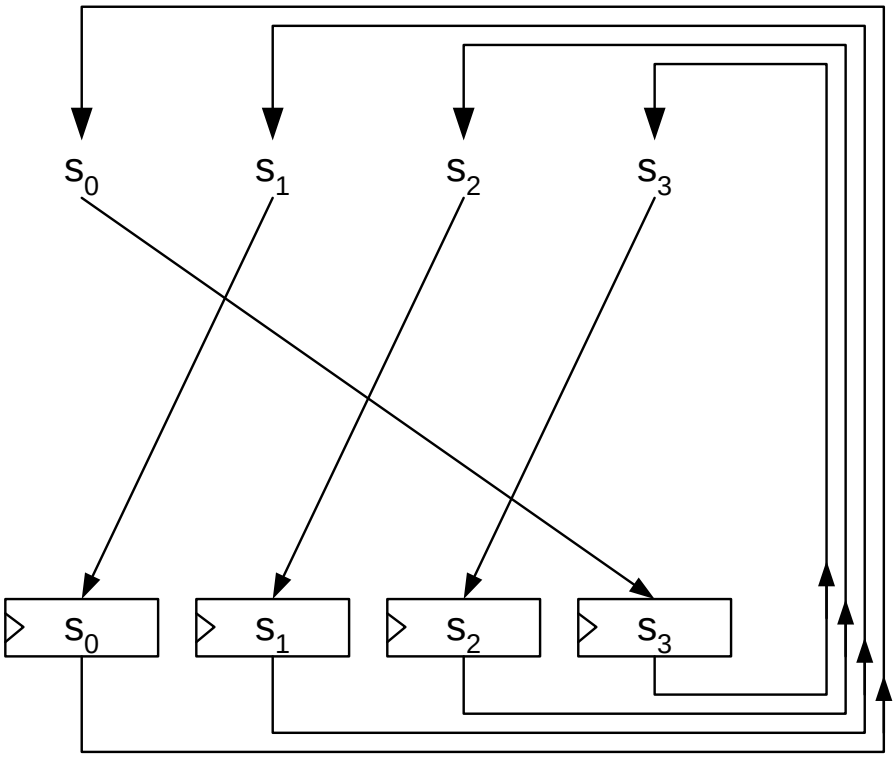
Specification of LUT-SR RNGs

- Describe LUT-SR RNGS using (*n*,*r*,*t*,*k*,*s*)
  - (*n*,*r*,*t*,*k*) describes the ***class*** of LUT-SR RNG
  - Free parameter *s* identifies an ***instance*** of the RNG class
- Provides a way to compactly describe RNGs
  - For example: (*n*=1024,*r*=32,*t*=5,*k*=32,*s*=7240)
  - Describes a LUT-SR generator with:
    - A period of 2^{1024}-1, producing 32 random output bits per cycle
    - Designed for 5-input XOR gates and 32-bit shift registers
- Where does *s* come from?
  - Identifies the best instances, details in paper

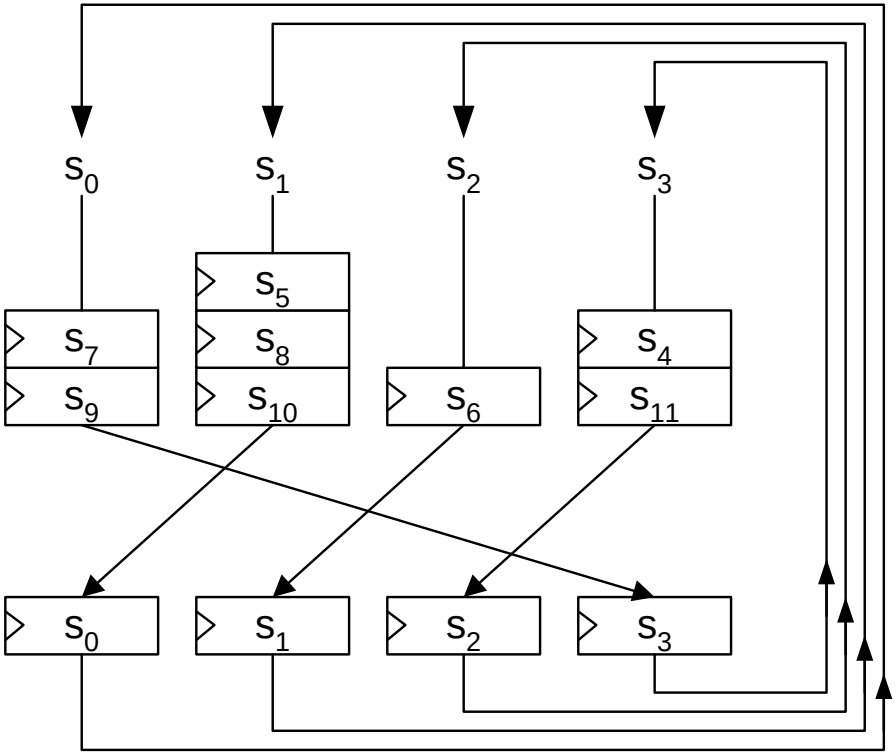
Expand 5 parameters into RTL

1. Seed simple (software) RNG with free parameter *s*
2. Randomly extend FIFOs
   - While state bits of RNG is less than *n*
     - Choose a random output bit *i* in [0,*r*)
     - If fifo_length(*i*) < *k* then extend fifo(*i*)
3. Create the shift permutation *i*=*i*+1 mod *r*
   - This will form our loading cycle
4. Insert XOR connections
   - For *i* in [1,*t*)
     1. Create a random permutation of [0,*r*) -> [0,*r*)
     2. Add permutation as XOR connections
5. Permute the output bits

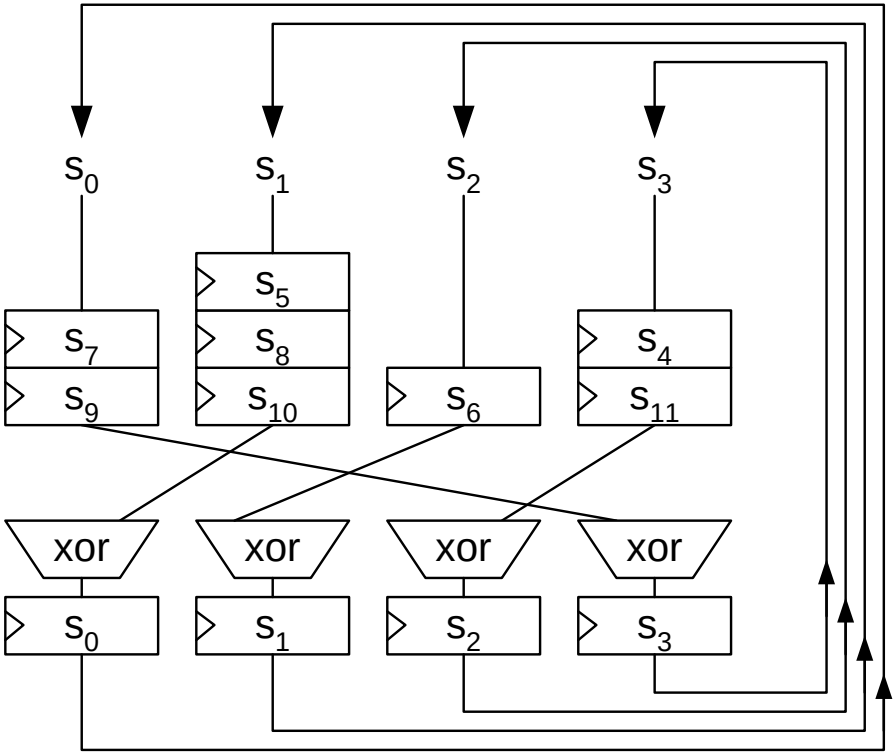
1. Initialise RNG using *s*
2. Create shift cycle



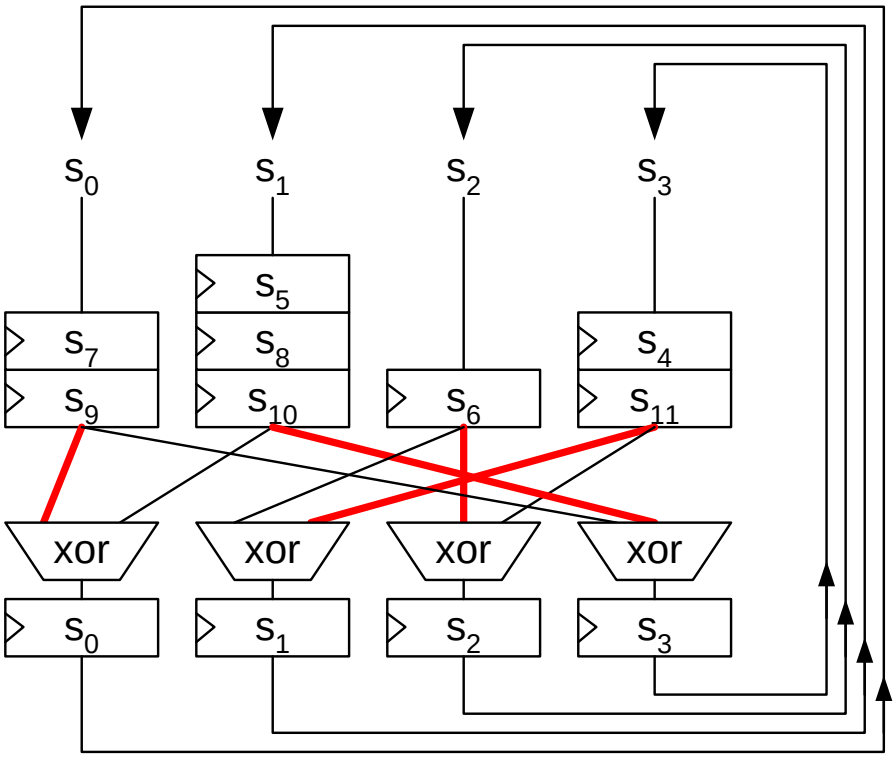
1. Initialise RNG using *s*
2. Create shift cycle
3. Randomly extend FIFOs



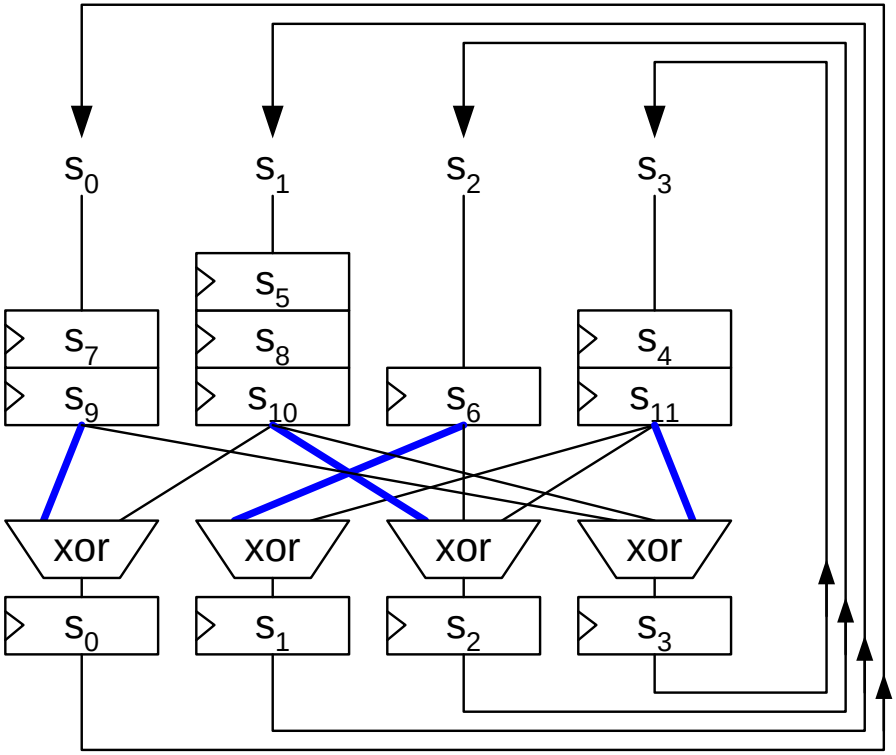
1. Initialise RNG using *s*
2. Create shift cycle
3. Randomly extend FIFOs
4. Build XOR connections



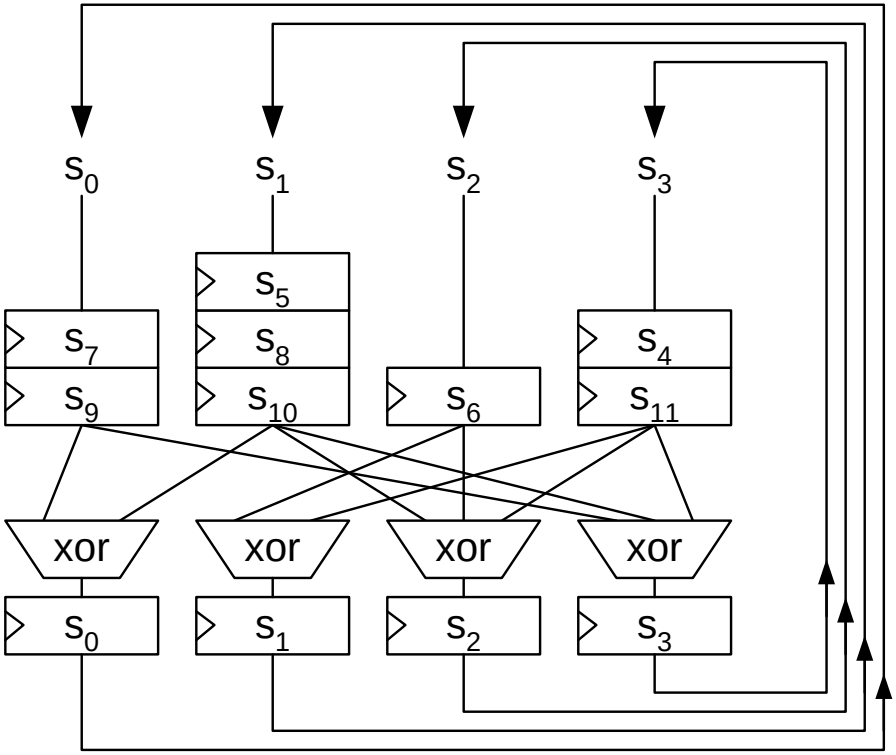
1. Initialise RNG using *s*
2. Create shift cycle
3. Randomly extend FIFOs
4. Build XOR connections
   - Random permutations



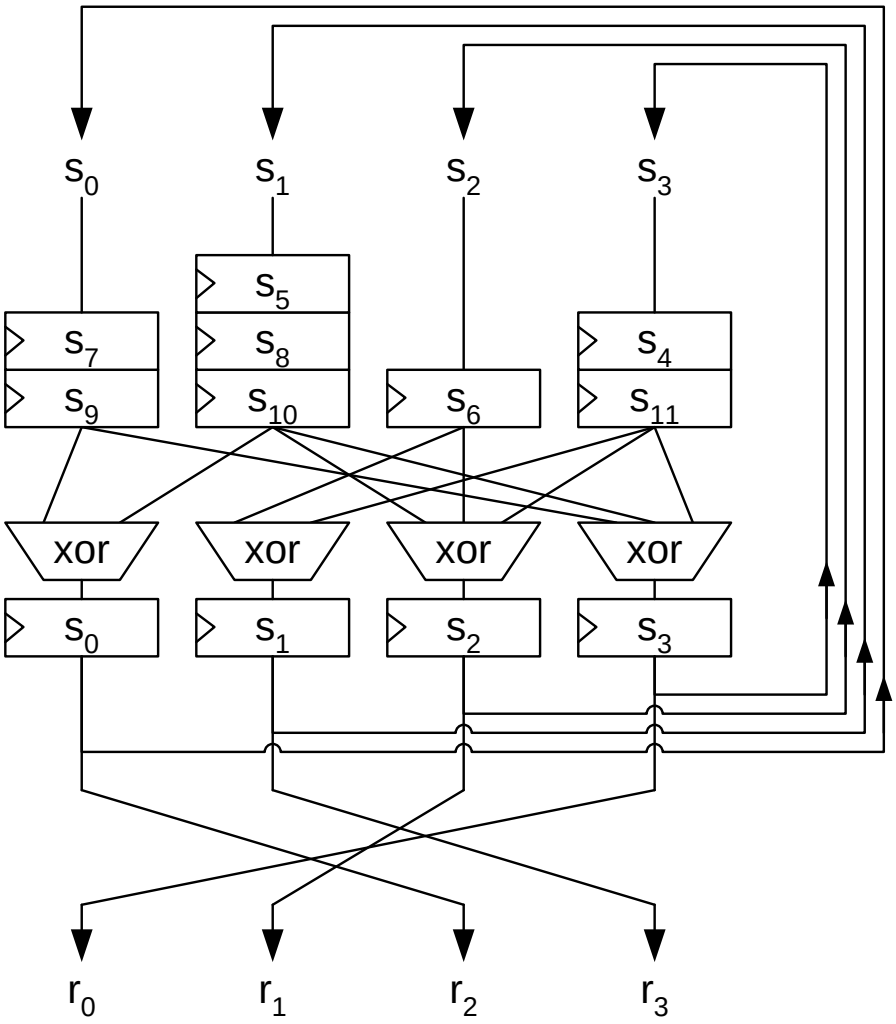
1. Initialise RNG using *s*
2. Create shift cycle
3. Randomly extend FIFOs
4. Build XOR connections
   - Random permutations
   - Keep existing edges



1. Initialise RNG using *s*
2. Create shift cycle
3. Randomly extend FIFOs
4. Build XOR connections
   - Random permutations
   - Keep existing edges



1. Initialise RNG using *s*
2. Create shift cycle
3. Randomly extend FIFOs
4. Build XOR connections
   - Random permutations
   - Keep existing edges
5. Permute output bits



Why write the algorithm this way?

- Not the most efficient algorithm for ***finding*** RNGs
  - Could modify chances of each *s* giving a valid RNG
  - Algorithm is quite slow when building up FIFOs

- But very efficient algorithm for ***instantiating*** RNGs
  - Code is very small and easy to understand
  - Self-contained and can be copied and pasted

- Algorithm design and spec. is biased towards users
  - There are many potential users (instantiators) of RNGs
  - There is only one person, me, doing the searching

Complete specification in the paper

- Provide everything in one column of C++
  1. Expansion algorithm: convert 5-tuple into RNG
  2. Pseudo-RTL source-code printer
     - Dump the RNG in a form that can be turned into VHDL or C
       - Sigh, or Verilog if you hate type-safety ***that*** much...
  3. Bit-accurate software simulator
- Paper includes tuples for a variety of *r* and *t*
  - Hopefully covers useful spectrum for most people
- Plus: online open-source repository
  - Generator for VHDL RNGs and test-benches
  - http://www.doc.ic.ac.uk/~dt10/research/rngs-fpga-lut_sr.html
  - Or try googling “LUT-SR RNG”, or look in paper for the URL

Comparison with other RNGs

	n	r	Quality	RAM	LUT	FF	r/LUT
Tauss113	113	32	Medium	0	87	208	0.37
MT19937	19937	32	Good	2	278	?	0.12
LFSR-160	160	32	Poor	0	448	384	0.07
LUT-OPT	512	512	Medium	0	513	512	1.00
LUT-FIFO	11213	89	Good	1	115	181	0.77
LUT-SR	1024	32	Good	0	64	64	0.50

Conclusion

- New approach for producing optimised RNG
  - Linear recurrence theory: ensure statistical quality
  - Compact RNG description using 5 parameters
- The LUT-SR generator is optimised for FPGAs
  - Bit-wise LUTs to improve mixing within state
  - Bit-wise SRs to increase period using cheap storage
- Provides a good balance between quality and area
  - High performance, only two LUTs per random bit
  - Provides long periods and great statistical quality
- Easy to use: VHDL available, or use the paper spec.
  - Friends don’t let friends use LFSRs...