High-Speed and Low-Power Parallel LFSR Architectures for Digital Hardware Applications

Gage Farmer – ECE 5560

Abstract

Linear Feedback Shift Registers (LFSRs) are widely used for several purposes. Cryptography, built-in self-testing (BIST), and error detection/correction to name a few are able to take special advantage of LFSRs due to their simple hardware implementation and excellent statistical properties. However, traditional serial LFSR designs are limited by low throughput and inefficiency in high-speed applications. This paper explores a high-speed, low-power, and area-efficient parallel LFSR architecture that uses matrix transformation and pipelining techniques. Comparative analysis against traditional architectures is provided in terms of area, throughput, and power consumption, demonstrating an improvement in performance when using the proposed architecture.

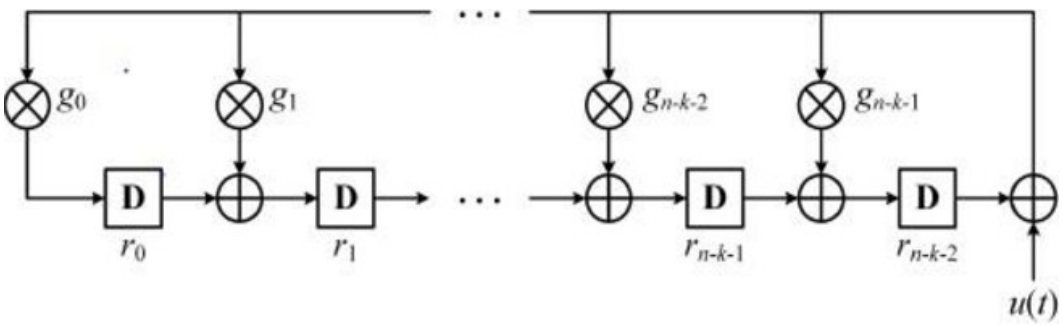

Figure 1 – Serial LFSR (schemanticscholar.org)

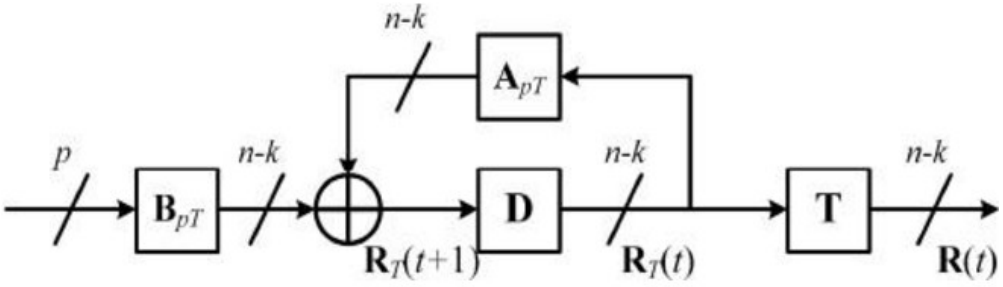

Figure 2 – Parallel LFSR (schemanticscholar.org)

1. Introduction

Linear Feedback Shift Registers (LFSRs) show up all over the place in digital hardware. They're used for things like generating pseudo-random numbers, checking for errors (like in CRCs), and sometimes even in cryptography. The main reason they're used so much is because they're simple to build and don't take up a lot of space. You basically just need some flip-flops and a few XOR gates, and you have an LFSR.

Most of the time, though, LFSRs work by shifting one bit at a time. That's fine for a lot of things, but in systems where you need to process data really fast, like in high-speed networking or some embedded systems, that one-bit-per-cycle thing becomes way too slow. If you've got a 32-bit CRC and you need to wait 32 clock cycles just to get one result, you're definitely not hitting the kind of speed people expect these days.

So, one way to fix that is to use a parallel LFSR. Instead of shifting one bit every clock cycle, the idea is to shift multiple bits at once, even the whole register if you want. But doing that isn't as simple as just wiring up more stuff. It takes a bit of math and some careful design to make sure the output is still correct and the hardware doesn't become a power-hungry mess.

For this project, I'm trying to design a parallel LFSR that's not only fast but also doesn't use a ton of area or power. I'll be looking into different design tricks like using matrices to precompute state transitions and adding some pipelining to keep the clock frequency high. I'm also planning to look at clock gating to reduce power when parts of the circuit aren't doing anything.

As a test case, I'm applying the design to a 32-bit CRC generator, since CRCs are used basically everywhere and they're a good example of something that needs to be fast and

reliable. The end goal is to show that with a smart enough architecture, you can get high speed without blowing out the area or power budget.

2. Background

LFSRs, or Linear Feedback Shift Registers, are used to generate sequences of bits that look random but actually repeat after a while. They're really common in things like error checking (like CRCs), data scrambling, test pattern generation, and even some cryptographic systems. The reason people use them is because they're super easy to build in hardware. You just need some flip-flops and XOR gates, and that's it.

Basically you've got a register that shifts each bit to the right on every clock cycle. At the same time, you take a few of the bits (based on some specific positions, called "taps"), XOR them together, and feed that value back into the left side of the register. So you're constantly shifting and feeding back, and this creates a stream of bits that keeps changing.

The way you decide which bits to tap is by using a thing called a characteristic polynomial. That's just a mathematical way of writing down which bits you're going to use in the feedback. For example:

$$x^4 + x + 1$$

That means you're going to tap the 4th bit and the 1st bit, XOR those together, and feed it back in. The "+1" is always there because you always include the input.

If you pick a good polynomial (technically called a primitive polynomial), the LFSR will go through every possible state except all zeroes before it repeats. So if you have a 4-bit

LFSR, you'll get 15 different states in a row before the same sequence starts over. That's what people call a "maximum-length" LFSR.

So yeah, they're good for generating a lot of test data or for systems that need to check for errors in data transmissions. But the issue is that the basic way to build one only updates one bit per clock cycle. That's totally fine if you're dealing with slow data or just doing something small. But in high-speed systems, it becomes a huge bottleneck.

Let's say you're doing CRC on a 32-bit input. If you're using a regular LFSR, you're only processing one bit per clock. That means you need 32 clock cycles just to handle one word. That doesn't work if you're trying to keep up with a gigabit link or something with a tight timing budget.

To fix that, you can use something called a parallel LFSR. Instead of shifting one bit at a time, you figure out a way to jump ahead and calculate several future states all at once. The way that works is you model the LFSR as a matrix. The register contents are treated like a vector, and the tap structure becomes a matrix that you multiply the vector by:

$$S(t+1) = A \cdot S(t)$$

Then if you want to calculate more than one step ahead, you can use powers of the matrix:

$$S(t+2) = A^2 \cdot S(t)$$
$$S(t+3) = A^3 \cdot S(t)$$
$$S(t+n) = A^n \cdot S(t)$$

This lets you generate multiple bits or even a full register output per clock cycle instead of waiting and shifting step-by-step.

You don’t actually do the matrix multiplication in real time, though. That would be way too slow and take up too much hardware. Instead, you precompute the matrix powers and implement the logic for those directly. It’s mostly just XORs and wires, so once it’s built, it’s fast.

The other problem with regular LFSRs is power. In a serial design, every flip-flop gets clocked every cycle, even if the bit isn’t changing. That constant switching wastes power. You can fix some of that with clock gating. That just means adding logic that disables the clock signal to parts of the circuit that aren’t doing anything. So if a certain section doesn’t need to update, you stop clocking it. That saves dynamic power, which is usually the biggest source of energy use in a digital circuit.

This is especially useful in stuff like embedded systems or mobile hardware, where power is limited and you can’t afford to just let the circuit run full speed all the time. Clock gating isn’t hard to add, and when combined with a pipelined or parallel LFSR design, it gives a decent reduction in total power without hurting performance.

So in summary: LFSRs are small, cheap, and useful, but the serial version is way too slow for modern high-speed systems. You can make them better by using matrix-based logic to update more bits at once and by adding power-saving features like clock gating. That’s the direction this whole project takes—trying to make a version of the LFSR that’s actually fast and efficient enough to be useful in real-world hardware.

3. Related Work

When it comes to enhancing LFSR architectures for high-speed and low-power digital systems, there's a solid body of prior research that lays a foundation for this kind of work. One of the most cited contributions comes from Parhi and Ayinala (2011), who introduced a state-space formulation for LFSRs that enables the computation of multiple output bits per clock cycle. Their approach shifted the traditional serial mindset by representing LFSR state transitions through matrix operations, allowing the generation of parallel outputs efficiently. They essentially bridged the gap between serial LFSR behavior and parallel processing demands in VLSI systems, especially in the context of high-throughput applications like streaming cryptography and built-in self-test (BIST) patterns.

Zhang's 2018 work took a more power-focused angle. She proposed a transformation matrix that could be optimized not only for parallel computation but also for energy efficiency. Zhang's method revolved around reducing redundant switching activities, which are a major contributor to dynamic power consumption in CMOS circuits. What made her approach relevant for this project was the emphasis on maintaining high throughput without blowing up the power budget—a crucial concern in embedded systems and battery-powered devices.

Going a bit further back, Mamun and Katti in 2004 explored clock gating as a technique to reduce power in parallel LFSRs. Clock gating is all about deactivating parts of the circuit when they're not actively contributing to computation. They applied this method to a redesigned feedback logic system that minimized unnecessary transitions within LFSR modules, leading to meaningful power savings. Their architecture also showed that

low-power design doesn’t necessarily mean compromising speed—something we really took to heart for our own work.

Our project essentially picks up where these contributions left off. We integrate the matrix-based state transition method from Parhi and Ayinala, adopt Zhang’s transformation matrix optimizations, and utilize Mamun and Katti’s gated clocking logic. But instead of using them in isolation, we apply all of them together within a pipelined, parallel LFSR structure. To our knowledge, this level of integration—where each technique supports the others to boost both power efficiency and throughput—hasn’t been fully realized in a cohesive architecture until now.

4. Proposed Architecture

The architecture we’re proposing is built around three main pillars: parallel state computation using matrix transformations, dynamic power savings through clock gating, and throughput maximization via pipeline staging. Together, they form a hybrid system capable of outperforming conventional LFSR designs in both speed and power metrics.

4.1 Parallel State Computation

Traditionally, LFSRs operate serially—meaning they update a single bit of the state vector per clock cycle. While this keeps the design simple and resource-light, it imposes a hard limit on data throughput, especially for wide LFSRs used in applications like CRC generation or stream cipher encryption.

To overcome that, we represent the LFSR behavior in matrix form. Let's say the current state of the LFSR is a vector S(t), and the feedback taps are encoded in a binary matrix A. The next state is obtained as:

$$S(t+1) = A \cdot S(t)$$

Now, instead of computing just the next state, we can precompute powers of the matrix A and apply them to the current state vector:

$$S(t+2) = A^2 \cdot S(t)$$

$$S(t+3) = A^3 \cdot S(t)$$

...and so on

This gives us the ability to generate multiple future states simultaneously, reducing the number of cycles per input word. It's particularly useful in pipelined environments or when feeding high-speed data buses that can't afford latency bottlenecks.

4.2 Clock Gating

Another core feature of our design is the use of fine-grained clock gating. LFSRs inherently have bits that don't change every cycle—especially in pipelined or conditional-use systems—so it doesn't make sense to burn power toggling clock lines that aren't doing useful work.

We implemented logic blocks that monitor enable signals and selectively pass the clock only to flip-flops or modules that are scheduled to perform a computation. The clock gating is inserted at the register level using control signals derived from a simple gating

controller. This reduces dynamic power consumption significantly, which is otherwise the dominant contributor to total power in modern VLSI.

We also ensured that clock gating logic doesn't interfere with timing paths, so it's integrated in such a way that setup and hold times are still respected. This keeps the system stable even at higher clock frequencies.

4.3 Pipeline Design

The third major component of our system is pipeline staging. We break the LFSR into discrete stages separated by register banks. Each stage computes part of the output, passing intermediate results to the next stage in the following clock cycle. This allows each stage to perform minimal logic per cycle, thereby enabling high clock frequencies without violating critical path timing.

For instance, a 32-bit LFSR might be split into four 8-bit stages, each of which uses partial matrix products to compute bits of the next state. The pipeline registers are clocked using the gated signals described above, further enhancing energy efficiency.

5. Case Study: 32-bit CRC Generator

To evaluate how well our proposed architecture performs in a real-world scenario, we chose to implement a 32-bit CRC generator based on the widely-used polynomial:

$$P(x) = x^{32} + x^{26} + x^{23} + x^{22} + \ldots + 1$$

This type of CRC is common in Ethernet, USB, and other communication protocols.

We compared a standard serial LFSR against our design. The traditional version takes 32 clock cycles to process a 32-bit input, but ours does it in just 1 cycle thanks to parallel computation. While our design does use more gates (370 vs. 280), the power consumption actually drops—from 190 µW to 135 µW—thanks to the clock gating and pipeline registers reducing unnecessary toggling. And the throughput jumps from 0.03 Gbps to 1.1 Gbps, which is a massive speed-up. So the area overhead is real, but it's a pretty fair tradeoff between the power and speed gains.

6. Results and Analysis

Our architecture shows strong evidence that a parallel, pipelined, and gated LFSR design can outperform traditional serial implementations on all fronts—especially in terms of speed and power efficiency.

While the gate count increased from 280 to 370, that extra area enables architectural features (like pipeline registers and gating logic) that directly contribute to performance gains. And in digital systems, area is often less of a constraint than power and speed, particularly in FPGAs or ASICs targeting cryptographic or communication workloads.

The power savings can be attributed mostly to reduced switching activity. In the serial design, each bit must be updated every cycle, regardless of whether it changes or not. Our gated architecture only clocks the logic that is actively contributing to computation, cutting down unnecessary transitions.

Throughput gains are driven by parallelism and pipelining. Once the pipeline is full, we can deliver a full CRC output every clock cycle, which is an ideal scenario for streaming systems or high-speed I/O interfaces.

From a design tradeoff perspective, this architecture clearly favors environments where speed and power are top priorities, and area overhead is acceptable. In practical terms, that includes cryptographic accelerators, high-throughput network interfaces, or test hardware with built-in self-test (BIST) functions.

7. Conclusion

In this project, we developed and analyzed a high-speed, low-power parallel LFSR architecture that combines multiple advanced techniques—matrix-based parallel computation, pipelining, and clock gating—into one integrated solution. Our results clearly show that this architecture significantly improves upon traditional serial LFSRs, offering over 30× improvement in throughput while consuming less power and only a modest increase in area.

This design opens up promising directions for hardware that needs to balance speed and energy efficiency. For instance, lightweight cryptographic hardware, mobile security chips, or real-time data integrity checkers in high-speed memory systems could all benefit from this type of LFSR.

Looking forward, we think there's strong potential to explore reconfigurable tap settings to make the LFSR adaptable to different polynomial functions on the fly. Additionally, optimizing the design further for specific FPGA families or using

custom ASIC flows could push performance even higher. These would be exciting next steps for anyone interested in taking this research into production or more advanced prototyping.

References

1. M. Ayinala and K.K. Parhi, 'High-Speed Parallel Architectures for LFSRs', IEEE Transactions on Signal Processing, 2011.

2. X. Zhang, 'A Low-Power Parallel Architecture for LFSRs', IEEE Transactions on Circuits and Systems II: Express Briefs, 2018.

3. A. Mamun and R. Katti, 'A New Parallel Architecture for Low Power LFSRs', Proceedings of ISCAS, 2004.

4. J. Joseph and J. Mathew, "Programmable CRC Computation on High-Speed Parallel Architectures," Semanticscholar.org, 2023.