Linear Feedback Shift Register

Francesco Iemma

Accademic Year 2020/21

Contents

Chapter 1

Introduction

The Linear Feedback Shift Register (LFSR) is a shift register [1] in which some bits are manipulated and fed back to the input in order to generate, thanks to the shifting, a sequence of bits. So we can say that its input is the output of a linear function of its previous state.
The initial value of the LFSR is the so-called seed and it affects the stream of bits generated by the LFSR. However, this stream is not (obviusly) really random because the number of possible states is finite and the stream of bits is determined by its current or previous state and, moreover, is possible to compute the bit generated knowing the seed and the feedback function. Anyway if the feeback function is well-choosen then the LFSR can produce a sequence of bits that appears random and which has a very long cycle, so this means that the output starts to repeat the generated numbers after a very long time. So we can assume, at least within a cycle, that it generates random numbers. For this reason the sequences generated from an LFSR are called pseudo-random.

1.1 Algorythm Description

In order to understand the LFSR algorythm we need to discuss about the *feedback function*: it is the function which describes the way in which some spefic bits of the shift register are manipulated and then fed back to the input. Usually the arrangment of the inputs of the feedback function is represented as polynomial mod 2. In our case the feedback polynomial is:

$$1 + x^{11} + x^{13} + x^{14} + x^{16}$$

Analysing the polynomial we can understand how LFSR works. In particular the feedback polynomial indicates which bits of the shift register affects the next state, these are called *taps*. In most cases the way in which the taps affect the next state is using XOR gates (but also XNOR gates can be used).
Now let's assume that we have an LFSR with a length of N and see how it works. At the starting state in the LFSR we have the seed. Then the feedback logic starts to work and the bit released from this logic is the new 0^{th} bit and the others bits are shifted by one to the right and due to the shift the previous last bit, i.e. the $(N-1)^{th}$ which has become[2] the N^{th}, will be the output of the LFSR.
In a more formal way (let's assume b'_i the i-th old bit, b_i the i-th new bit, N the length of LFSR with $N > 0$, T the set which contains the indexes of the taps and y the function which represents the feedback logic) I can say:

- $b_0 = y(b'_j, ..., b'_k)$ where $j, ..., k \in T$
- $b_i = b'_{i-1}$ for $1 \leq i < N$
- $output = b'_{N-1}$

[1] A shift register is a sequential logic circuit made up of chain of flip flops, for each rising edge of the clock the value of each flip-flop is shifted to the next flip-flop

[2] with N^{th} we indicate the bit which is out of the LFSR. A shift register with a length of N has N bits and so its bits are the ones in the interval [0, N-1], the N^{th} doesn't exist but it's a way to indicate the "overflow" bit.

The taps in our case are the bits 11, 13, 14, 16 (we can recognize them gazing to the exponents of x in the feedback polynomial) so this means that the inputs of the feedback logic are these. And so in our previous notation $T = \{11, 13, 14, 16\}$, then $b_0 = y(11, 13, 14, 16)$.
In the following image it's possible to see a logic representation of a generic LFSR.

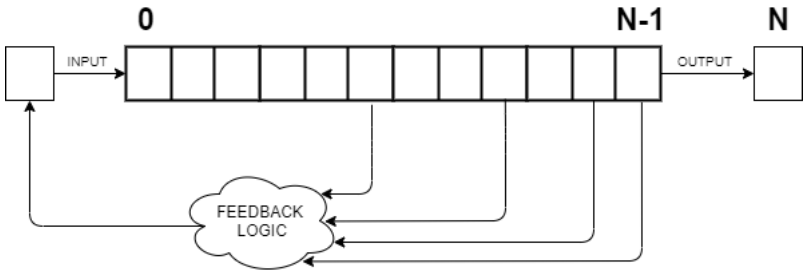

Figure 1.1: Logic representation of a LFSR

1.2 LFSR's Properties

In order to complete this brief journey through the world of LFSRs, let focus on a particular type of LFSR which has an interesting property, this is the so-called maximum-length LFSR. An LFSR is a maximum-length one, if and only if, the feedback polynomial is primitive and so it's necessary that:

- The number of taps is even
- The taps values are coprime

The property of a maximum-length LFSR is the following: if its length is N then it will assume all the possible $2^N - 1$ states (the state with all 0 is excluded because otherwise it will stop).
Others properties for generic LFSRs are:

- As we can saw previously, the output is deterministic, in fact it's possible to compute it starting to the current state
- The state with all zeroes is not allowed, for this reason even a maximum-length LFSR cannot generate a sequence of 2^N states

It's important to point out that even if the output is deterministic the LFSRs are spreadly used, this happens because some techniques can be adopted to resolve this issue, for example giving an irregular clock to the device or manipulating the output stream with some non-linear combination of two or more bits extracted from this stream.

1.3 Possible Applications

LFSRs are spreadly used in a lot of fields. Let's see the most important ones:

- They are used in Criptograhpy as pseudo-random number generators for stream-chipers (especially in military applications). In this field it's very important to resolve the issue about the deterministic behaviour of the LFSR in order to avoid possible attacks, some possible strategies are the ones seen in the previous section.
- They are used in Communication, for example in the Global Position System (GPS) and also in radio-jamming in order to generate a pseudo-random noise. Then another application in this field is for scrabbling, the bits produced from an LFSR are combined with the data bit in order to have a convenient stream of data that ensures good properties and better performance from the viewpoint of the modulator/demodulator. Then it is also used in CRC (Cyclic Redundancy Check).
- They are used in Electronics in order to do a pseudo-random testing of a circuit in order to test "all" possibile inputs of the circuit (in fact this technique is also called pseudo-exhaustive testing)

1.4 Possible Architectures

There are two type of possible LFSR's architectures:

- Fibonacci LFSR (also called "Many-To-One architecture", because from "many" bits you obtain "one" bit that is the new input bit)
- Galois LFSR (also called "One-To-Many architecture", because from "one" bit - that is the last one and also the output - you obtain "many" bits that are the ones which are in positions next to the taps)

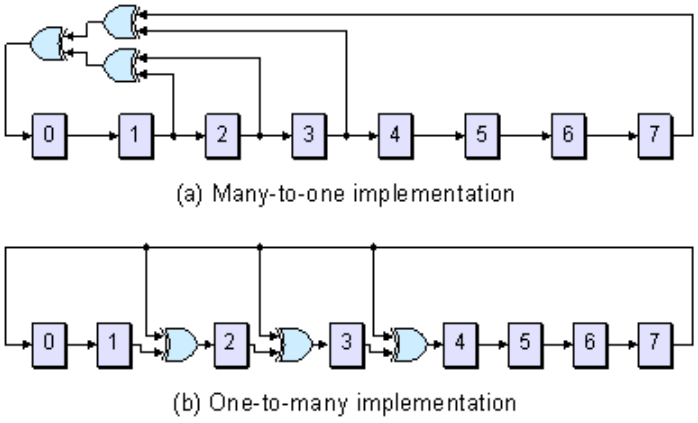

Figure 1.2: LFSR Architectures - Source: www.eetimes.com

1.4.1 Fibonacci LFSR

The Fibonacci LFSR is the simplier possible architecture for an LFSR and it is also the standard one, basically the generical things said so far about LFSRs describe the Fibonacci LFSR. In fact, for instance, we can observe that the figure 1.2.(a) and the figure 1.1 are indeed the same figure with the difference that the 1.1 is a logic representation and so the gates have not been drawn (they are in the so-called "Feedback Logic").

1.4.2 Galois LFSR

The Galois LFSR is an alternative architecture for LFSR, from the point of view of the output it's equivalent to Fibonacci one (the output of a Galois it's the same of a Fibonacci) but indeed they are internally different. On the rising edge of the clock, all bits that are not taps are shifted unchanged to the right by one position. Instead the taps are XORed with the previous last bit and the output of this XOR operation goes to the next bit on the right. In a more formal way we can write (let's assume b'_i the i-th old bit, b_i the new i-th bit, N the length of LFSR with $N > 0$, T the set which contains the indexes of the taps):

- $b_0 = b'_{N-1}$
- $b_{i+1} = b'_i$ with $i \notin T$, $0 \leq i < N-1$
- $b_{j+1} = XOR(b'_{N-1}, b_j)$ with $j \in T$, $0 \leq j < N-1$
- $output = b'_{N-1}$

1.4.3 Comparison

The Galois architecture has a big advantage with respect to Fibonacci one. In fact in the latter there is a chain of XOR gates and the XOR operations must be done in serial and it cannot be done in parallel. Furthermore, the propagation time of this chain is not negligible. On the other hand, in the Galois implementation, the XOR operations can be done in parallel

and the propagation time regard a single gate (not a chain as in the Fibonacci's case). So the propagation time of Galois implementation is lesser than the Fibonacci one, this means that with this architecture we can achieve a faster clock frequency. In our case we have four taps, so if we assume that we have only XOR gates with two inputs and each one has a $t_{propagation_i} = x$ we are in the following situation:

- Fibonacci: we have to use three XOR gates (installed as we can see in the following image) and so the minimum clock period T is equal to $T = t_{propagation_1} + t_{propagation_2} = 2x$. Then the maximum frequency achievable is $f = 1/T = 1/2x$. If $x = 4ns$ then $f = 125MHz$

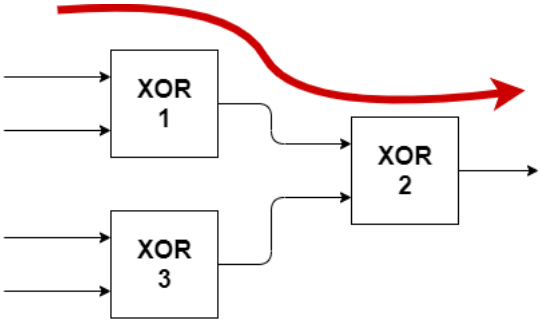

Figure 1.3: Critical path (in red) related to XOR gate in Fibonacci with 4 taps

- Galois: also in this case we use three XOR gates, but in this case each one is installed between two flip-flop so the critical path is made up of one gate. Then $T = t_{propagation_1} = x$ and so $f = 1/T = 1/x$. If $x = 4ns$ then $f = 250MHz$.

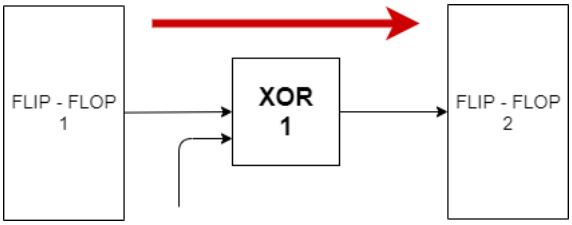

Figure 1.4: Critical path (in red) related to XOR gate in Galois

We can see that in this case $f_{Galois} = 2f_{Fibonacci}$ but if the number of taps increase we have a larger difference, for example with $NumTaps = 8$ I have $f_{Galois} = 3f_{Fibonacci}$, with $NumTaps = 16$ then $f_{Galois} = 4f_{Fibonacci}$. So if $NumTaps$ is a 2^a then $f_{Galois} = af_{Fibonacci}$. This results shown as the Galois implementation is better from the viewpoint of the maximum clock frequency.
This difference is bigger when the number of taps grows, in most application this difference is not so evident as in the example shown and, at the end of the day, the performance are more or less the same. For example if we use XOR gate with more inputs than two the $t_{propagation}$ of Fibonacci is less than the one in the case of a cascade of XOR with two inputs. So the relation with Galois is not so unbalanced. Then in some FPGA are used LUTs that support 6 input logic function, so using a XOR with two inputs or a XOR with six is basically the same.
Indeed, theoretically speaking, the difference in performance between the two implementations exists, and so in the following report it will be treaten the implementation in VHDL of an LFSR with Galois architecture and then its testing and its practical realization obtained by programming the ZyBo Board with Xilinx Vivado.

Chapter 2

Architecture Description

As we discussed in the previous chapter, we will implement a Galois LFSR. Let's see the necessary devices to implement it. In order to implement a memory element I need a D Flip-Flop with set and reset. In fact we need to initialize each memory element to the seed value, so the i-th flip flop will be initialize with the i-th bit of the seed. The aim of the implementation is to have a functional LFSR, but another aim is also having a customizable LFSR. In order to do this we have to add a circuit to change (or not) the input of a simple flip-flop, we call this circuit *tap circuit*.

2.1 D Flip-Flop with tap circuit

Assume that we have the i-th flip-flop and the j-th (with $j = i+1$). If i is a tap then its output must be XORed with the $N-1$ bit, otherwise the output of i goes unchanged to the input of j. So we have to implement the following logic function (described using a pseudo-code and assuming that $feedback$ is the $N-1$ bit):

```
void logicFunctionOmega(isTap_i, feedback)
{
        if(isTap_i==true)
                input_j = xor(output_i,feedback);
        else
                input_j = output_i;
}
```

Now we have to translate this logic function from this pseudo-code to logic gates. But, before doing this, we need to point out some concepts:

- $isTap_i$, $feedback$ and $output_i$ are bits. We will use the following convention: $isTap_i$ is true if it is 1, it is false when it is 0.
- We know that the neutral element for the XOR is the 0 (if A and B are the inputs of XOR, if B is 0 then A passes the gate unchanged), in fact the truth table for the XOR is:

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.1: XOR Truth Table

After fixed this points we can implement our unknown function, let's call it Ω and let see what is the expected truth table (in the image A is $isTap_i$ and B is $feedback$):

A	B	AΩB
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.2: Ω Truth Table.

Remember that the output must be equal to B ($feedback$) if A(tap_i) is equal to 1, otherwise it must be equal to the neutral element of the xor, so 0.
We can easily recognize that the "mysterious" logic function Ω is indeed the AND gate. So the logic block diagram of a flip-flop with the tap circuit is the following: