Nonlinear feedback shift registers and generating of binary de Bruijn sequences

Christian Ebne Vivelid

November 21, 2016

Master's thesis

Department of Informatics

University of Bergen

Introduction

Cryptology is the science of hiding the meaning of a message by concealing it, so that it is hard to find the meaning for others than the intended receiver. There are mainly two ways of doing this, one is to use different words in the message instead of the real one. This is called using *code words.* The other method is called using *cipher* and is performed by changing the individual characters of the message in an orderly way. This process is called encryption, and the process of retrieving the original message is called decryption.

The original message is called the *plaintext* and the encrypted *plaintext* is called *ciphertext. Ciphers* can be further divided into two types of *cipher*s, *stream ciphers* and *block ciphers. Block ciphers* takes a number of plain text characters and a key to produce the *cipher* text. While *stream ciphers* a uses a key and some kind of generator to produce a pseudo random sequence of character that is uses with the *plaintext*, to produce the *ciphertext.*

Stream ciphers

A sequences is a number of consecutive characters. E.g. a number *1563* is a sequences of the characters 1,5,6,3. And the word *cipher* is a sequence of the characters c,i,p,h,e, r. The simplest sequences are those that only contains only two different characters, we call such a sequence binary. The notation a “bit” is used to refer to either a 0 or a 1, and all binary sequences can be expressed by bits by swapping the two original characters for 1 and 0. It is a well known fact that every sequence can be represented by a sequence of bits, by having a number of subsequent bit represent one of the characters of the original sequence.

The binary sequence generated by a generator and a key is called a *key stream,* and is a pseudo binary random sequence. Then a *plaintext* can be encrypted be performed an operation called *exclusive or* between the *plaintext* and the *key stream.* This operation is that bit number i of the *ciphertex*t is 1, if bit number i of the *plaintext* and the *key stream* are different bit (e.g. 0 and 1). If they are the same (e.g. 0 and 0) then it is 0. The *ciphertext* can then be sent through an insecure channel, and the receiver can decrypt the *ciphertext* by generating the same *key stream* and use the *exclusive or* operation between them to produce the *plaintext.* This process is illustrated in figure 1 below.

Any *cipher* must have a good security, and while it is generally hard to clearly specify the criteria for this. There are some known ways to attack *stream ciphers* based on some properties. So if the *cipher* does not have these properties, it should be relatively secure. One of these properties is randomness distribution. If there is a clear and simple pattern to the sequence, it is quite easy to break it. Or if the sequence can be found by simply guessing or using a small part of the sequence to find the rest.

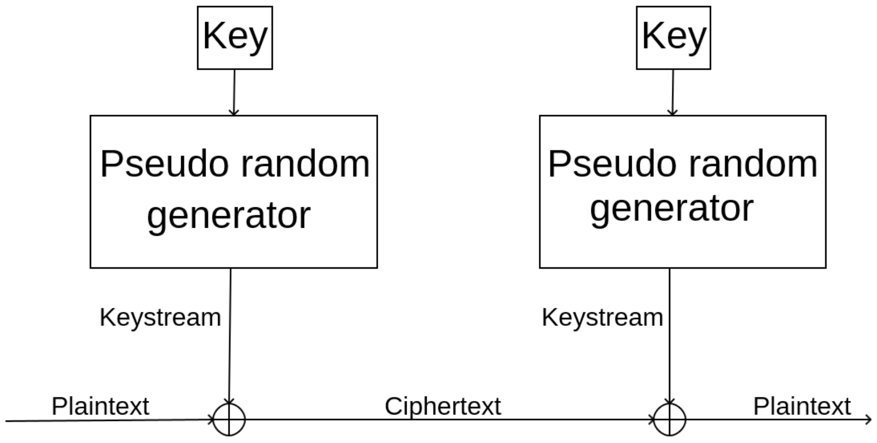


Figure 1: *Stream cipher* model

In this thesis we will look at a way to construct a generator that produce a *key stream* with a high period and complexity. But first we will have to look at a type of generators that is commonly used in *stream ciphers*, it is a device called a feedback shift register (*FSR*).

There is two types of these, *Linear feedback shift register* (*LFSR*) is one of them. The theory of *LFSR* is largely complete and it is easy to generate long sequences with many good properties from *LFSR*s. But they are easily recognized, so they have a relatively low security. The other type is *Nonlinear feedback shift register* (*NLFSR*) that can be very hard to recognize, but theory of *NLFSR* is largely incomplete. The problem of generating a special type of sequences from *NLFSR* called de Bruijn sequences be the main focus of these thesis.

First the two things that determine the generated sequence of a *FSR* is the feedback function and the initial state. The feedback function is a Boolean function that determine the next bit of the sequence based on some of the previous bits of the sequence. The initial state is the first n bits of the sequence, that is set at the start.

Sequence have 3 important properties, which is period, complexity and distribution. Period is the number of bits of the sequence before it starts to repeat itself. Complexity is a measurement on how many bit is needed to find the sequences from a part of the sequence and some other information. Distribution is the difference of the number of 1's and 0's in the sequence. Ideally the distribution should be zero and the complexity the whole of the sequence.

Overview for this thesis

The main purposes of this thesis is to give a light run through of generally *FSR,* and a thorough look at *NLFSR* and de Bruijn sequences. There properties and ways to generating them. Finally we will present a method to construct de Bruijn sequences and a hardware implementation of this method. That uses $O(log_2^2 n)$ operations and a total of $O(n \cdot log_2 n)$ components to generate a bit of a de Bruijn sequence of order n from another one of order m.

This method generates 2^{n-m} uniquely different de Bruijn sequences, depending on a chosen binary sequence of length $(n - m)$, where every different sequence correspond to a unique de Bruijn sequence of order n. For every de Bruijn sequence of order m the set of 2^{n-m} unique de Bruijn sequences of order n, have no common elements with any other of the sets. The number of de Bruijn sequences of order m is $2^{2^{m-1}-m}$, and the total number of de Bruijn sequences of order n that can be generated by this method from all de Bruijn sequences of order m is $2^{n-m} \cdot 2^{2^{m-1}-m} = 2^{2^{m-1}+n-2\cdot m}$.

In part 1 we will look at the general properties of *FSRs* and *LFSR.* Part 2 continues by looking at the basic properties of *NLFSR* and part 3 contains more advanced topics about *NLFSR.* Part 4 details the known algorithms to find de Bruijn sequences and finally part 5 present a method to construct de Bruijn by using the inverse D-homomorphism and a hardware implementation of this method is presented.

1 *FSR*

The theory of *LFSR* is largely complete, and it is possible to deduce the composition and properties of any *LFSR* [20], [24] and [11]. They also share some of the same properties with *NLFSR*s and since *LFSR*s can be used to create *NLFSR* with specific properties, a short overview about them will be given in this section.

There are two types of *FSR*, Galois and Fibonacci. We will mainly be looking at Fibonacci *FSR* through most of this thesis and only briefly touch upon *Galois FSR* in the latter half. A feedback shift register (*FSR*) consist of a number of memory cell, that each contain a binary bit (either 1 or 0).

The cells are numbered from 0 until $(n-1)$, and *FSR*s are usually divided into different sub sets depending on the number of memory cells. Which can be referred to as number or length/size, but in this thesis the term order will be used to refer to the number of memory cells of a *FSR*.

We will use the notation $(x_0, x_1, x_2, ..., x_{n-1})$ to refer to the binary values of these cells and call this the state of the *FSR*. x_i refers to the value binary value of memory cell i and call x_i one of the state variable of the *FSR*. The state variables of a *FSR* is referred to as the state of the *FSR*. The values of the memory cells changes every time a signal called the clock signal is sent to *FSR*. We call this processes to *"clock"* the *FSR*. For the $n-1$ first cells, the cell value for cell i changes to that of cell $i+1$:

$$x_i = x_{i+1} \ : 0 \leq i \leq n-2$$

For cell number $n-1$, its new value is determine by a Boolean function f that depend on the current state of the *FSR*:

$$x_{n-1} = f(x_0, x_1, ..., x_{n-1})$$

The output sequence from an *FSR* can be any one of the memory cells, one can take one bit (e.g. x_0) for every clock of the *FSR* and then this is the sequence of the output bits of the *FSR*. This sequence is determent by the Boolean function and the initial state of the *FSR*. The initial state of a *FSR* is the value of memory cells at the start.

1.1 *LFSR* and Sequences

Cycles and period

A cyclic shift of a sequence is the original sequence where all the entries of the sequence have been moved to the entry on the left from the original position. Except for the first entry, that is instead moved to the last entry.

Definition 1. Given a sequence $S = (s_0, s_1, ..., s_{n-1})$, a single cyclic shift of this sequences is $E^1(S) = (s_1, s_2, ..., s_{n-1}, s_0)$. A ith cyclic shift $E^i(S) = (s_i, s_{i+1}, ..., s_{i-2}, s_{i-1})$.

The period p of a sequences S is the smallest positive integer, such that:

$$E^p(S) = (s_p, s_{p+1}, ..., s_{p-2}, s_{p-1}) = S$$

A sequence S can be shorten if $p < n$, to a cycle $C = (s_0, s_1, ..., s_{p-1})$. The original sequence S can be reconstructed from C by treating$C$ n times. Use the term "cycles/sequences generated by a *FSR*" to refer to the set of all the possible cycle/sequences generate by a *FSR*. All cyclic shift of a sequence /cycle will not be treated as different sequences/cycles as the structure is identical, and it is simple to derive all cyclic shift of a sequence/cycle of a sequence/cycle.

ANF and recursive Boolean function

The algebraic normal form (ANF) of a Boolean function f: $\{0,1\}^n \rightarrow \{0,1\}$ is an n-variable polynomial in $GF(2)$ on the form:

$$f(x_0, ..., x_{n-1}) = \Sigma_{i=0}^{2^n-1} c_i \cdot x_0^{i_0} \cdot x_1^{i_1} \cdot ... \cdot x_{n-1}^{i_{n-1}}$$

$c_i \in \{0,1\}$ and $(i_0 i_1 ... i_{n-1})$ is the binary expression of i with i_0 being the most significant bit and i_{n-1} being the least significant. This form is used for this thesis and it may be presume that any function is on ANF form if nothing else is stated.

The total number of all possible Boolean functions is 2^{2^n}, of these only 2^n are *LFSR*. A Boolean function can be simplified by introducing sub functions, that is a function $f(x_0, x_1, ...)$ may be expressed as $f = h(x_0, x_1, ...) + g(x_0, x_1, ...)$. Here f is expressed by the two sub functions h and g. Define *term* as any sub function that does not contain any addition modulo 2. E.g. given a Boolean function $f(x_0, x_1, x_2) = x_1 \cdot x_2 \oplus x_0 \oplus 1$, then x_1, 1 and $x_1 \cdot x_2$ are all the terms of f. Note that a term is a sub function, but not all sub functions are terms.

Characteristic polynomial of a *LFSR*

All *LFSRs* have a linear feedback function, meaning that all parts of the function can only depend on up to one of the n state variables. So a *LFSR* can have any of the variables x_0 until x_{n-1} as a term of its feedback. But it cannot contains any term on the form $x_i \cdot ... \cdot x_j$, $0 \leq i, j \leq n-1$. So there is $n+1$ possible terms, that can be a part of the feedback function. Then all possible combinations sum up to:

$$\Sigma_{i=0}^{n+1} \left(\begin{array}{c} n+1 \\ i \end{array} \right) = 2^{n+1}$$

A characteristic polynomial is an other way to represent a *LFSR* and it is in a one-to-one and onto relation to linear feedback functions. Meaning that every linear feedback function have a unique characteristic polynomial and vice versa. Below in figure 2 is a *LFSR* with the feedback function $f(x_0, x_1, ..., x_{n-1}) = x_0 \oplus x_{n-2} \oplus x_{n-1}$ and this corresponds to a the characteristic polynomial $f(x) = x^n + x^{n-1} + 1$.

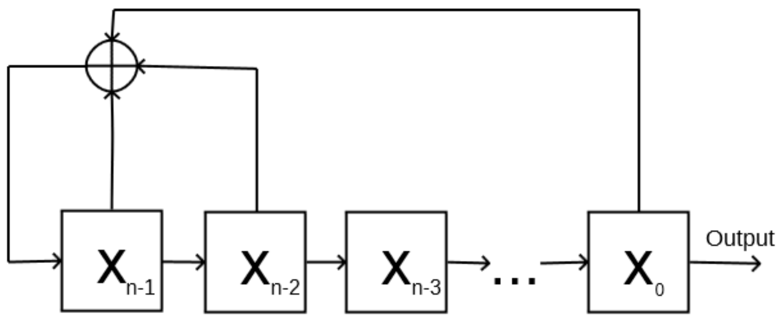


Figure 2: A *LFSR*

The relation between a characteristic polynomial and the corresponding linear feedback function is as follows:

$$f(x) = c_1 \cdot x^n + c_2 \cdot x^{n-1} + ... + c_n \cdot x + c_0 \Longleftrightarrow$$
$$f(x_0, x_1, ..., x_{n-1}) = x_0^{c_0} \oplus x_1^{c_1} \oplus ... \oplus x_{n-1}^{c_{n-1}} \oplus c_n$$

Good graph

The Good graph (also called de Bruijn graph) is a graph over 2^n vertexes, where each vertex represent one of the possible states of a *FSR* with order of n. All the vertexes have two outbound edges and two inbound edges. The outbound edges represent the register changing its state to one of the two possible next state, and the inbound edges represent the two possible previous state.

So any sub graph of this that satisfy the condition that all vertexes have only one outbound edge, correspond to a *FSR*. The notation B_n is used to refer to the Good graph over 2^n vertexes or for *FSRs* of order n. A *FSR* of order n is said to be in B_n. Below in figure 3 is the Good graph for order 3, B_3.

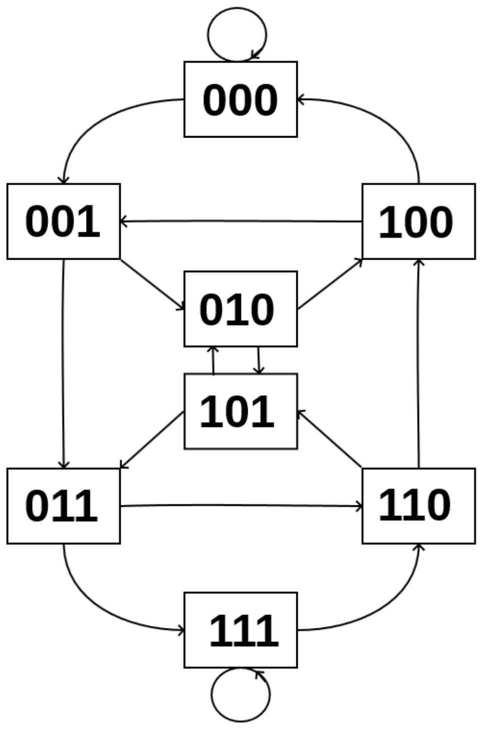


Figure 3: Good graph for B_3

Singular and non-singular

A *FSR* with a feedback function f is called singular if there is at least one pair of conjugate states $S = (s_0, s_1, ..., s_{n-1})$ and $Con(S) = (\overline{s}_0, s_1, ..., s_{n-1})$ such that $f(S)) = f(Con(S))$. So there will be at least one state $(s_1, ..., s_{n-1}, f(S))$ that has two predecessors.

This is illustrated under in the graph on the right, where the state $(1, 0, 1)$ have two predecessors $(0, 1, 0)$ and $(1, 1, 0)$. If there is no such pair, then the *FSR* is called *non-singular.* All states will then only have one unique predecessor and successor. So if the function is *non-singular,* then all initial states of the *FSR* will generate a cycle.

An illustration of this is shown in the graph below on the left. That is the de Good graph for $f(x_0, x_1, x_2) = x_0 \oplus x_1 \oplus x_1x_2$, that generates the cycles $(0, 0, 0)$, $(0, 1, 0, 1, 1, 0)$ and $(1, 1, 1)$. Any sub graph of the Good graph that satisfy the condition that all vertexes have only one outbound edge and inbound edge, correspond to a non-singular *FSR.*

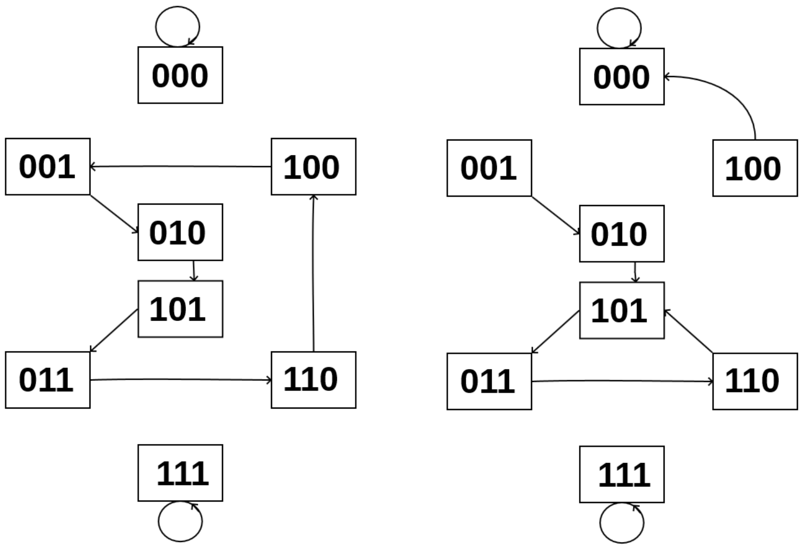


Figure 4: Singular and non-singular sub graphs of B_3

Theorem 1. *A FSR is non-singular if and only if its recursive feedback function f is on the form $f(x_0, x_1, ..., x_{n-1}) = x_0 \oplus g(x_1, ..., x_{n-1})$.*

For a *FSR* to be non-singular any to conjugate must have unique successors, but the only difference is the first state variable x_0. So the feedback function must depend on x_0 for it to be non-singular. All the conjugate states successors have to be unique, so the term x_0 have to be independent of the rest of the terms. As this will make it impossible for $f(S) = f(Comp(S))$ because the difference in the first bit will complement the result of the recursive function.

1.2 Maximum-length *LFSR* and M-sequences

LFSRs can be divided into two specific set, the set of all *LFSR* that can generate a maximum-length sequence of $2^n - 1$ and the set of the rest of *LFSR*s. $\frac{\phi(2^n-1)}{n}$ *LFSRs* of order n can generate a maximum-length of $2^n - 1$(ϕ is Eulers totient function).

Finding a *LFSR* that generate a maximum sequence is relatively simple, because a maximum-length *LFSR* will have a characteristic polynomial that is primitive. And a primitive polynomial have to be irreducible. A polynomial $p(x)$ is irreducible if and only if:

$$p(x) = f(x) \cdot g(x),\ g(x) \neq f(x)\, and\, 1 \leq degree(g(x)), degree(f(x)) \quad (2)$$

An irreducible polynomial $p(x)$ is primitive if:

$$x^{(2^n-1)/q_i} \neq 1\, mod(p(x)),\ \text{for all prime factors } q_i \text{ of } 2^n - 1 \quad (3)$$

So it is relatively easy to check if a polynomial is primitive, and the probability of randomly selecting a primitive polynomial by chance is $\frac{2^n}{\frac{\phi(2^n-1)}{n}} \approx \frac{1}{n}$ if no additional analyze is performed. So as long as n is not to big, the process to find a *LFSR* that can generate a maximum-length sequence is relatively easy.

The sequence generated by such a *maximum-length LFSR* is called an M-sequence. They have some useful and presumed unique properties, that give the M-sequence good pseudo randomness. We will examine some of these in the next sub sections.

Golombs randomness postulate

All sequences generated by any *FSR* is predetermined, but this can make them predictable. We need way to determine if a generated sequence will look random, that there is no clear overall pattern to the sequence generated. The Golumbs randomness postulates are 3 rules that ensure that any sequence that follows them have a good pseudo randomness. That they will look and have relatively the same properties as a truly random sequence of bits, while still being predetermined.

Definition 2. A *Run* is a number of consecutive 0's or 1's. A *Block* is a *Run* of 1's. A *Gap* is a *Run* of 0's.

Postulate 1. The number of zeros and number of ones differ by at most one during a period of the sequence.
Postulate 2. Half of the *Runs* in a full cycle have length 1, one $\frac{1}{4}$ of all *Runs* have length 2, $\frac{1}{8}$ have length 3 etc, as long as the number of *Runs* exceed one. Moreover, for each of these length there are equally many *Gaps* and *Blocks*.
Postulate 3. The out of phase *auto-correlation* of the sequence always has the same value.

All M-sequences obey and are the model for these postulates.
This 3 conditions provides that pseudo randomness is guarantied and therefore is usable for simulating true randomness. In the next section we will look at the property correlation and a special case of correlation named auto-correlation.

Correlation

Correlation is a measurement of the similarity of two phenomena. In this thesis correlation is used on binary sequences, but it can also be used to measure all kinds of quantifiable phenomena. Given two sequences S_1 and S_2 of length l, the correlation between them is an integer on the interval $-l$ to l. Where a correlation of l means that the sequences are identical and $-l$ that they are each others complement. If it is 0, then there is no bias.

The correlation between two sequence $C(S_1, S_2) = l - 2 \cdot D(S_1, S_2)$, where $D(S_1, S_2)$ is the number of sequence bit of S_1 and S_2 that are different. E.g. the two sequences $S_1 = (1, 0, 0, 1, 1)$ and $S_2 = (0, 0, 1, 1, 0)$ have a correlation $C(S_1, S_2) = 5 - 2 \cdot D(S_1, S_2) = 5 - 2 \cdot 3 = -1$.

Auto-correlation of a sequence S is the correlation between the sequence and a cyclic shift of itself. Denote this by $C_S(i) = C(S, E^i(S))$, and there is l possible auto-correlation values for the l unique shift. One of the most interesting property of a sequence is the maximum auto-correlation of a sequence except for $C_m(0)$

The auto correlation of an M-sequence m of order n is:

$$C_m(0) = 2^n - 1$$

$$C_m(\mathrm{t}) = -1, \text{ if } t \neq 0 \, mod(2^n - 1)$$

So M-sequences satisfies the 3rd of Golumbs randomness postulate.

Linear complexity

Definition 3. The linear complexity of a binary sequence is the minimal possible order for a *LFSR* that can generate it. Denoted $L(S)$ for the linear complexity of the sequence S.

For any sequence S generated by a *LFSR*, at most $2 \cdot L(S)$ bits of the sequence is needed to find a *LFSR* that can generate the whole sequence by using the Berlekamp-Massey algorithm [17]. That gives a minimal *LFSR* that will generate the entire sequence. This means that the security of a depend on the linear complexity of the key stream. In addition to other properties of the key stream, that also have to be sufficient enough to provide the desired level of security.

Berlekamp-Massey algorithm

Given a binary sequence $S = (s_0, s_1, ..., s_{n-1})$, and the nth discrepancy $d_n$is:

$$d_n = s_n \oplus c_1 s_{n-1} \oplus ... \oplus c_L s_{n-L}$$

The following algorithm produces a characteristic polynomial f_S that generates the sequence S.

1. If $S = (0, 0, ..., 0)$, then $L(S) = 0$ and $f_S = 1$. If $S = (0, 0, ..., 0, 1)$, then $L(S) = n + 1$ and f_S can be any *LFSR* of length n.
2. Else, if $d_n = 0$, then $L_n = L_{n-1}$and $f_n = f_{n-1}$.
3. If $d_n = 1$, then:

$$L_n = \{ \begin{array}{c} L_{n-1},\, L_n > N/2 \\ n - L_{n-1},\, L_n \leq N/2 \end{array}$$

Let m be the largest integer such that $L_m < L_{n-1}$.

$$f_n = \{ \begin{array}{c} f_n + x^{2*L_{n-1}-n-2} + f_m,\, if\, L_{n-1} > N/2 \\ x^{n-2*L_{n-1}} + f_m,\, if\, L_{n-1} \leq N/2 \end{array}$$

it is also possible to solve the problem of finding the generating polynomial $P(x)$ for a sequence of linear complexity L by solving a system of linear equation. With a time complexity of $O(n^2)$, that can be further optimized to $O(n \cdot log_2 n)$. This is shown in matrix form below in figure 5.

$$\begin{bmatrix} s_0 & s_1 & \cdots & s_{n-1} \\ s_1 & s_2 & \cdots & s_n \\ \cdots & \cdots & \cdots & \cdots \\ s_{n-1} & s_n & \cdots & s_{2n-2} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \cdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} s_n \\ s_{n+1} \\ \cdots \\ s_{2n-1} \end{bmatrix}$$

Figure 5: Matrix for finding the generating polynomial for a sequence

Cycle structure

Any non-singular Boolean function f will generate a number of cycle, that may be of different length.

Definition 4. Two Boolean function f_1 and f_2 of the same order have the same cycle structure if and only if they have the same number of cycles of length i, for all i.

An example of cycle structure is de Bruijn sequences. It is also possible to define more specific structure, that have more requirements in addition to cyclic structure. E.g. maximum-length *LFSR* all have the same cycle structure and have the cycle of all zeroes.

The are methods to determine the cyclic structure of *LFSR*s, based on the properties of its characteristic polynomial. As this polynomial can be polynomial reduced to its irreducible polynomial factors and based on this properties can be deduced. E.g. if the set of irreducible polynomial factors contains some multiple factors $f(x)$, it is possible to find the cycles generated by $f(x)^k$ from looking at the cycles of $f(x)$. For any two polynomial $f(x)$,$h(x)$ it is possible to find the cycles generated by $f(x) \cdot h(x)$ from looking at the cycles generated by $f(x)$ and $h(x)$. So it is possible to find the cycles for any *LFSR* form its polynomial irreducible factors.

2 Basic properties of *NLFSRs*

A nonlinear feedback shift register is a *FSR* which feedback function contains at least one term that is a factor of multiple state variables (e.g. $x_1 \cdot x_2$). This increases the total number of possible feedback function when compered to *LFSR*, and it is also makes it possible to have linear complexity higher than n. Which is the upper limit for the linear complexity of a sequence generated by a *LFSR* of order n, while sequences generated by a *NLFSR* of order n have an upper bound of $2^n - 1$. In the figure below is a *NLFSR* with the recursive function $x_{n-1} = x_o \oplus x_{n-1} \oplus x_{n-2} x_{n-3}$, that can also be expressed by a feedback function $f(x_0, x_1, ..., x_{n-1}) = x_o \oplus x_{n-1} \oplus x_{n-2} \otimes x_{n-3}$

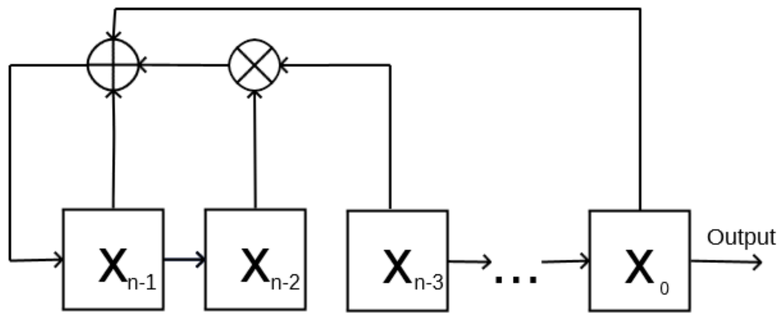


Figure 6: A *NLFSR*

2.1 Cycle joining and splitting

Pure cycling register

The pure cyclic register (PCR) is the non-singular function $f(x_0, ..., x_{n-1}) = x_0 \oplus g(x_1, ..., x_{n-1})$ of any order n and $g(x_1, ..., x_{n-1}) = 0$. Then the weight of g's truth table is 0, as all the 2^n entries of the truth table are 0. Then the number of cycles of f is equal to

$$Z(n) = \frac{1}{n} \cdot \Sigma_{d|n} \phi(d) \cdot 2^{n/d}$$

and the total number of cycles for a function of order n can not be greater than $Z(n)$. This was conjectured by Golomb [11] and proven by Mykkeltveit [19]. Based on this and the fact that changing one of the entries in the truth table of g for any non-singular *FSR* f, leads to an increase or decrease of one in the number cycles generated by f.

Since the weight of $g's$ truth table for PCR is 0, and the number of cycles of PCR is an even number as $Z(n)$ is even for all n. All the other non-singular functions can be created by incrementally changing the entries of g. So for any non-singular function f, the parity of the weight of g's truth table is equal to the parity of the number of cycles generated by f.

For the function $f(x_0, ..., x_{n-1}) = x_0 \oplus g(x_1, ..., x_{n-1})$ of any order n and $g(x_1, ..., x_{n-1}) = 1$. Then the weight of f is 2^n as all the 2^n entries of the truth table are 1. Then the number of cycles of f is equal to

$$Z^*(n) = \frac{1}{2n} \cdot \sum_{\substack{d|n \\ d:odd}} \phi(d) \cdot 2^{n/d}$$

This is proven in [9].

Cycle joining and splitting

Given a non-singular feedback function $f(x_0, x_1, .., x_{n-1}) = x_0 \oplus g(x_1, .., x_{n-1})$. By changing one of the entries in the truth table of g, the successor of two states is swapped. For a state $a_1 = (1, \lambda)$ and its successor $a_2 = (\lambda, c).b_1 = (0, \lambda)$ and its successor $b_2 = (\lambda, \overline{c})$, λ is some entry in the truth table of g (e.g. the all zero state $(0, 0, 0, ..., 0)$) and $c \in \{1, 0\}$. Then by complementing the λ entry of g, the successor of a_1 changes to b_2 and the successor of b_1 changes to a_2. This can be used to join two cycles, if the two states a_1 and b_1 are on different cycles. If they are on the same cycle, the cycle will split in two cycles.

Below in figure 7 this is illustrated by going from the left figure to the right, which shows two cycles being joined into one. And from the right figure to the left one, shown a cycle being split into two.

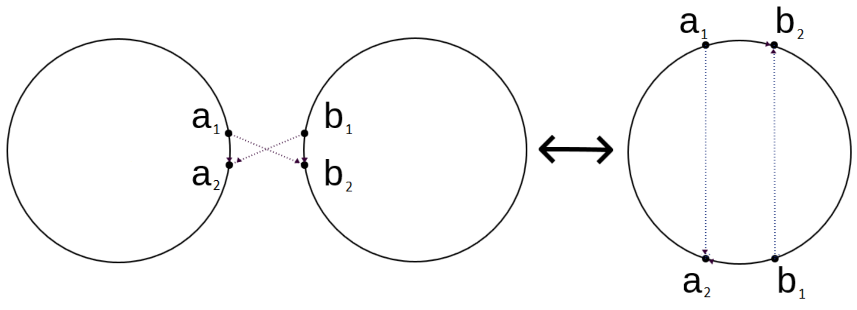


Figure 7: Cycle joining and splitting

2.2 Cross-join pairs

Definition 5. Cross-join pairs are sets of two pairs of conjugate state $\{a, \hat{a}\}$and $\{b, \hat{b}\}$ on a cycle C. Such that interchanging the successors of $\{a, \hat{a}\}$or $\{b, \hat{b}\}$ splits C into two cycles C_1 and C_2. Where $\{b, \hat{b}\}$ or $\{a, \hat{a}\}$ are on different cycles, and interchanging the successor of the pair that was not used to split C will join C_1 and C_2 into a new cycle D.

The order of interchanging the conjugate pair does not matter, as interchanging the successor of a conjugate pair correspond to changing one of the entries in $g's$ truth table. Where the feedback function generating C is $f(x_0, ..., x_{n-1}) = x_0 \oplus g(x_1, ..., x_{n-1})$, and the order of changing two entries in $g's$ truth table is commutative as long as C is a binary cycle.

The number of cross-join pairs for an M-sequence was proven to $\frac{(2^{n-1}-1)(2^{n-1}-2)}{6}$ in [12].

2.3 De Bruijn sequences

A de Bruijn sequence is a sequence over an alphabet of k-symbol with a period of k^n where all possible n-tuples appears. It was rediscovered by de Bruijn [2] and generalized for larger alphabet, the binary de Bruijn sequences was original discovered by Flye Sainte-Marie [8].

In Cryptology most work with de Bruijn sequences are over a binary alphabet, as this correspond to binary computers. Biology and other fields of science can use larger alphabet. The main focus will be on binary de Bruijn of length 2^n, will use the formulation "de Bruijn sequences of order n" to refer to a de Bruijn sequence of period 2^n.There is a total of $2^{2^{n-1}-n}$ different de Bruijn sequences of order n.

Properties

Chan, Games and Key [3] proved that the linear complexity of de Bruijn sequences S of order n is bound by $2^{n-1}+n+1 \leq L(S) \leq 2^n - 2$. Also it is impossible for a de Bruijn to have a linear complexity of $2^{n-1}+n+1$. The complement of a de Bruijn sequence S will have the same complexity as S [3].

Requirements

It is possible to deduce some simple requirement for an ANF feedback function of order n that generates a de Bruijn sequence of length 2^n. The function will have to be non-singular as there cannot be any multiple predecessors for any state. Because this makes it impossible for the generated sequence to be of maximum-length. Since the feedback function only have one cycle, the all zero state $(0, 0, 0, ..., 0)$ of the *FSR* must not give a 0. This means that it have to be 1. Else the all zero state would be a part of a cycle with a length of one, since it would be its own predecessor and successor. This is contradiction, as the total number of cycles have to be one.

Next the parity of the sub function g is equivalent to the parity of the cycles generated by a non-singular *FSR*. This means that the parity g will have be odd, for any feedback function f that generates a de Bruijn sequence.

Lemma 1. *The only term that changes the parity of the truth table for the sub function g for any non-singular function f for order n, is the term $x_1x_2...x_{n-1}$.*

Proof. Denote the original feedback sub function as g. The constant term 1 complement all 2^{n-1} entries in truth table of g, so the new weight of $g' = 2^{n-1} - weight(g)$. So if the parity of g is even it is even for g', and if the parity g is odd it is odd for g'. So the parity is the same as the original one.

Next by adding a term of j state variables changes 2^{n-1-j} entries of the truth table of g, $n-1 > j$. If the number of changes in the truth table from $1 \to 0, 0 \to 1$ is even, then the parity is same as the g. If the number of changes are odd, then changes$1 \to 0$ changes the parity of the truth table, and the changes $0 \to 1$ also changes the parity of the truth table. So the result of this is that the parity overall is unchanged. If $n-1 = j$ then the term of j state variables change only one entry. So this changes the parity of the truth table.

So any function that generates a de Bruijn sequence must have this term. □

Another property is that the weight of the truth table have to be in the interval:

$$Z(n) - 1 \leq W(f) \leq 2^n - Z^*(n) + 1$$

As it has been shown that the feedback function with the weight of g is 0 has $Z(n)$ cycles and any change of the parity of the truth table of g will change the parity of the of the number of cycles [11],[19]. So if the weight of g is incremented the number of cycles changes by one. The minimum weight of g of a de Bruijn function have to be at least $Z(n) - 1$.

Another restriction can be found from looking at the non-singular Boolean function f with the weight of g equal to 2^n, as this function generates $Z*(n)$ cycles and an decrementation of the weight of g will change the number of cycles by one. The weight of any de Bruijn function has to be at most $2^n - Z^*(n) + 1$.

Finally since a function of order n that generates a de Bruijn sequence of order n, only have one cycle. The all one state $(1,1,1,...,1)$ of the *FSR* must have the state $(1,1,...,1,0)$ as its successor, so $f(1,1,1,...,1) = 0$. As all nonzero terms will give a 1 for the state (1,1,1,...,1), the total number of terms most be even for this to happen.

These facts sum up to a general form that all ANF feedback function that generates a de Bruijn sequence of the same order will have to be on:

$$f(x_0, x_1, ..., x_{n-1}) = x_0 \oplus g(x_1, ..., x_{n-1})$$

$$g(x_1, ..., x_{n-1}) = 1 \oplus x_1x_2...x_{n-1} \oplus ....$$

$$Weight(terms(g(x_1, ..., x_{n-1}))) \equiv 0 \, mod(2)$$

The number of non-singular *FSR* is $2^{2^{n-1}}$, and half of these functions have the constant term 1. Since this is a requirement for a de Bruijn function, the number of candidates for *FSR*s that can generate de Bruijn of order n is $2^{2^{n-1}-1}$. Half of these will have the term $x_1x_2...x_{n-1}$, that is also a requirement to a de Bruijn function.

This limits the number of candidates to $2^{2^{n-1}-2}$. Since half of these again will have an even weight of gs truth table, the final numbers of candidates is $2^{2^{n-1}-3}$. Which makes it possible to find all the de Bruijn functions for order 3, without doing any testing.

Maximum-length ***LFSR*** vs maximum-length ***NLFSR***

The main differences between maximum-length *LFSR* and maximum-length *NLFSR* is illustrated below in table 1, that shows the most important differences between them. When $n \geq 3$, they have the following properties.

	LFSR	*NLFSR*
Number of functions	$\frac{\phi(2^n-1)}{n}$	$2^{2^{n-1}-n}$
Length	$2^n - 1$	2^n
Maximum linear complexity	n	$2^n - 1$
minimum linear complexity	n	$2^{n-1} + n$
Maximum number of terms	$n + 1$	2^n

Table 1: The differences between maximum *LFSR* and *NLFSR*

2.4 Representation

Any *FSR* of order n can be represented by a Boolean feedback function of the n state variables. From the n state variables it is possible to construct 2^n logically different ANF terms. Where $(n + 1)$ are the linear terms that consist of one of the n state variable and the constant 1, that can be used to construct *LFSR*s. Any implementation of a *NLFSR* will have to do some operations before the next bit from the feedback function can be calculated.

First all the different terms have to be checked if they are true or not. Then it will have to check if the number of true terms is even or odd. The complexity of finding the truth values of the different terms, depend on the number of state variables for each term. This can be performed by looking up all the state variables that the term depends on.

Now let T denote the number of terms of the feedback function f. To find if the output of f is true or false for any state, it depends on if the number of true terms is even or odd. So all $t \in T$ will have to be checked to find the truth value of the function. This may be relative slow, if the number of terms is relatively large.

In part 5, a hardware implementation is shown that can generate a bit of a de Bruijn sequence of order n in $O(log_2^2 n)$ basic operations. With a memory and component complexity of $O(n \cdot log_2 n)$. In part 2 we saw that a *LFSR* can be represented by a characteristic polynomial and that the property of this polynomial allowed us to deduce information about the period and cycle structure of the *LFSR*.

For any implementation there is a number of factors that can reduces the efficiency of generating bits. In software this is measured in time and memory and in hardware it is time and memory/components. For a *NLFSR* this can be further specify into 3 values that measure the efficiency of the implementation.

The time to clock a bit

One of the most important property of any representation for a *FSR*, as this can directly limit the usability. If it cannot produce the required amount of bits per time unit, it is not usable.

The time to switch one state to another

This time depend on the size of a state in the implementation, and how the process of changing from a state is implemented.

The size of a state

The size of all the information store in a single state of the *FSR* is usually equal to the number of memory cells. Of course there may be a difference between the order of the *FSR* and the linear complexity of the sequence generated by the *FSR*.

In the next section we will look at some advanced *NLFSR* sub classes and ways *NLFSR* can be used as building blocks to generate complex sequences.

3 Advanced properties of *NLFSRs*

3.1 Symmetric feedback function and Kjeldsen mapping

The symmetric shift registers has been studied by Kjeldsen [13] and Søreng [21],[22]. They have a minimal period dividing $n \cdot (n+1)$.

3.2 General algorithm

An algorithm is a clearly defined step-by-step list of operations that have to be performed to solve a task. The two main resources for an algorithm is time and memory. Time refers here to the amount of time or bound of time needed to run the algorithm. Memory is the maximum amount of required memory needed to run the algorithm.

Algorithm to find the feedback function from a truth table

The task of find the feedback function from a truth table, can be performed by the following algorithm. For a truth table for a *FSR* of order n, where the binary sequence $(t_0, t_1, t_3..., t_{2^n-1})$ represents the entries in the truth table. A state of the *FSR* is on the form $(s_0, s_1, s_2, ..., s_{2^n-1})$. t_0 is the truth value for the state $(0, 0, 0, ..., 0)$, t_1 is the truth value for the state $(0, 0, 0, ..., 1)$ and so on. Denote f_{temp} as the temporarily Boolean function. While $Bin(S)$ as the decimal value of the state S, where the leftmost bit is the most significant one and the rightmost is the least significant bit.

1. Start with $i = (0, 0, 0, ..., 1)$. If $t_0 = 1$, then $f_{temp} = 1$. Else $f_{temp} = 0$.
2. If $t_{Bin(i)} \neq f_{temp}(i)$, then $f_{temp} = f_{temp} + \prod_{s_i=1} x_i$. Increment i and repeat step 2 until $i = 2^n$.

This produces the feedback function on algebraic normal form (ANF).

3.3 Modified de Bruijn sequences

By removing the all zero state or the all one state from a de Bruijn sequence, a new type of sequence is created. Denote the first as a type 0 modified de Bruijn and the latter as a type 1. Given a de Bruijn sequence S, let m_0S denote the type 0 modified de Bruijn sequence created from S and m_1s the type 1. Let $DS(n)$ denote the set of all de Bruijn sequences of order n. $DS_0(n)$ denote the set of all type 0 modified de Bruijn sequences and $DS_1(s)$ the set of all type 1, both created from the elements of $DS(n)$. Note that the set of all M-sequences of order n, is a subset of $DS_0(n)$. Mayhew and Golomb [18] proved that for $n \geq 4$, the linear complexity of a type 0 sequence S is bounded by:

$$n \leq L(S) \leq 2^n - 2$$

An overview of the complexity distribution for orders 4,5 and 6 is also given. Zheng, Cao, Zhou and Xu [23] proves the following theorem:

Theorem 2. *If $s \in DS_0(n)$, $m_0s \in DS_0(n)$ and $m_1s \in DS_1(s)$, then $L(m_1cs) = LC(m_0s) + 1$, $L(m_0cs) = L(m_1s) - 1$.*

Which gives a complexity comparison between s, m_0s and m_1s for order 4 and 5. In [21] Mayhew gives a run through the auto-correlation of type 0 modified de Bruijn sequences, for orders 4, 5 and 6.

3.4 Galois *FSR*

A Galois *FSR* differs from a Fibonacci *FSR* by that all the memory cells of the *FSR* may have a feedback function. While the Fibonacci *FSR* only have a feedback function for the last cell. Because of this, the Galois *FSR* are more complex to analyze and the period of state of the register may be greater than the period of some of the memory cells. So the output sequence will depend on which of the memory cells is the output. As the output stream usually is one of memory cells. But as the state of the memory cells determines the next state, the period of the register cannot be greater than 2^n. Below in figure 8 is a Galois *FSR* with the feedback function $f_{n-1} = x_o \oplus x_{n-1} \oplus x_{n-2}$, $f_{n-2} = x_{n-1} \oplus x_{n-2}$ and $f_i = x_{i+1} \ : 0 \leq i \leq n-3$.

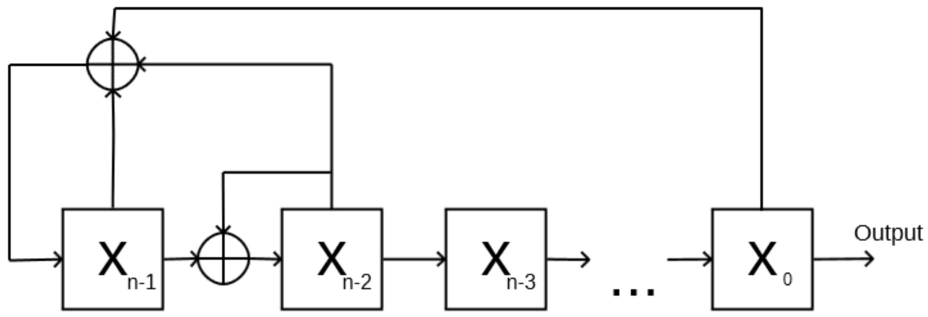


Figure 8: A Galois *FSR*

In the next part we will look at a transformation from a Fibonacci *NLFSR* to an equivalent Galois *NLFSR*.

3.5 Transformation from a Fibonacci to an equal Galois *NLFSR*

Dubrova [5] present a transformation from a Fibonacci *NLFSR* to an equivalent Galois *NLFSR*. This transformation is based on shifting the terms of the feedback functions.

Definition 6. Let f_a and f_b be feedback functions of the bits a and b of a *NLFSR* of order n. The operation shifting, denoted by $f_a \overset{P}{\to} f_b$, moves a set of product-terms $P \subseteq term(f_a)$ from the ANF of f_a to the ANF of f_b. The index of each variable x_i of each product-term in P changed to $x_{(i-a+b)}mod(n)$.

To make sure that this operation gives an equivalent *NLFSR*, some conditions have to be satisfied.

Definition 7. The terminal bit τ of a *NLFSR* of order n is the bit with the maximal index which satisfies the following condition:
For all bits i such that $i < \tau$, the feedback function f_i is of type $f_i = x_{i+1}$.

Define $min\_index(f)$ and $max\_index(f)$ as the smallest and largest index variable that the function f depends on. E.g. for $f = x_3 \oplus x_2 x_1$ $min\_index(f) = 1$ and $max\_index(f) = 3$.

Definition 8. An *NLFSR* of order n is *uniform* if:
a) All its feedback functions are singular functions of type (1).
b) For all its bits $i > \tau$, the following condition holds:
max_index(g_i)$\leq \tau$, where τ is the terminal bit of the *NLFSR*.

Theorem 3. *Given a uniform NLFSR with the terminal bit a, a shifting* $f_a \overset{P}{\to} f_b$, $P \subseteq term(f_a)$, $b < a$. *Results in an equivalent NLFSR if the transformed NLFSR is uniform as well.*

Algorithm to find a fully shifted *Galois NLFSR from a Fibonacci NLFSR*

Dubrova [5] also gives an algorithm to find the fully shifted Galois *NLFSR*. Given a uniform Fibonacci *NLFSR* N of order n, the fully shifted Galois *NLFSR* $\hat{N}$ which is equivalent to N is obtained as follows:

First, the terminal bit τ of $\hat{N}$ is computed as:

$$\tau = \max_{\forall p \in A_{g_{n-1}}} (max\_index(p) - max\_index(p))$$

Then, each product product-term $p \in A_{g_{n-1}}$ with $min\_index(p) \leq (n-1) - \tau$ is shifted to $g_{n-1-min-index(p)}$:

$$g_{n-1} \xrightarrow{\{p\}} g_{n-1-min-index(p)}$$

and each product product-term $p \in A_{g_{n-1}}$ with $min\_index(p) > (n-1) - \tau$ is shifted to g_τ:

$$g_{n-1} \xrightarrow{\{p\}} g_\tau$$

This provide a way to transform a Fibonacci *NLFSR* to an equivalent Galois *NLFSR*, and Dubrova [6] gives a survey of the improvement to efficiency this gives to *NLFSR* by using this transformation.

3.6 Combinators

By combing multiple systems like *LFSR* and *NLFSR*, it is possible to create new key stream generators. it is also possible to combine them with logic blocks to further create complex systems. We will give a brief overview of some different techniques to create Combinators.

Multiplication of sequences

The bitwise multiplication of sequences is an easy way to generate sequences with higher period and linear complexity. Given two repeating binary sequences S and T with $per(S)$, $per(T) > 0$. If $gcd(per(S), per(T)) = 1$ then the bitwise multiplication of S and T will produce a new sequence $S \otimes T$ with period $p(S \otimes T) = per(S) \cdot per(T)$ and the linear complexity $L(S \otimes T) = L(S) \cdot L(T)$.

Note that the distribution is generally bad, as the sequence will contains 3 times more zeroes than ones. So the possibility that a random bit of the sequence is 1 is 25 % and the possibility that a random bit of the sequence is zero is 75 %.Which makes it easy to use a correlation attack on the key stream, to find S and T.

3.7 Basic ways to create de Bruijn sequences from others of the same order

A few basic ways to find de Bruijn sequence from another one of the same order.

By reversing a de Bruijn sequence

Theorem 4. *The reverse of a de Bruijn sequence is also a de Bruijn sequence for any k-alphabet.*

Proof. A de Bruijn sequence of order n contains all possible state of length n. For any state $S = (s_0, s_1, ..., s_{n-1})$ of any de Bruijn sequence B, the reverse state $R(S) = (s_{n-1}, s_{n-2}, ..., s_0)$ is also a state of B as its a length is n. The reversing of a state is a one-to-one and onto homomorphism. So all k^n the states of the reversed sequence will be unique states, which is the definition of a de Bruijn sequence. □

For any non-singular feedback function $f(x_o, x_1, x_2, ..., x_{n-2}, x_{n-1})$ there exist a transformation to a Boolean function $f^{'}$ such that $f^{'}$generates all the reversed cycles of f. Given by $f^{'} = f(x_o, x_{n-1}, x_{n-2}, ..., x_2, x_1)$, so by simply swapping the index of state variables from 1 until $n - 1$ to produces a function that generate the reverse cycles of the original function. So this will generate the reverse de Bruijn sequence for any de Bruijn sequence.

By complementing a de Bruijn sequence

Theorem 5. *The complement of a de Bruijn sequence is also a de Bruijn sequence.*

Proof. A de Bruijn sequence of order n contains all the possible state of length n. For any state $S = (s_0, s_1, ..., s_{n-1})$ of any de Bruijn sequence B, the inverse state $\overline{S} = (\overline{s_0}, \overline{s}_1, ..., \overline{s}_{n-1}) = C(S)$ is also a state of B as its length is n. And the complement of a state is a one-to-one and onto homomorphism. So all k^n states of the inverse sequence will be unique states, which is the definition of a de Bruijn sequence. □

For any non-singular feedback function $f(x_o, x_1, x_2, ..., x_{n-2}, x_{n-1})$ there exist a transformation to a Boolean function $f^{'}$ such that $f^{'}$generates all the complement cycles of f. Given by $f^{'} = f(x_o \oplus 1, x_1 \oplus 1, x_2 \oplus 1, ..., x_{n-1} \oplus 1) \oplus 1$, so by simply complementing all the variables and the function itself produces a function that generate the complement cycles of the original function. So this will generate the complement de Bruijn sequence for any de Bruijn sequence.

By complementing the reverse of a de Bruijn sequence

Theorem 6. *The reverse complement of a de Bruijn sequence is equal to the complement reversed of de Bruijn sequence, and also a de Bruijn sequence.*

Proof. By the theorem 12 and 13, it is clear that both are de Bruijn sequences. The complement of any binary sequence is simply switching the symbols, so the structure of the sequence is identical. So the reversed complement is structurally equal to the reversed also a de Bruijn sequence and the reverse sequences of a de Bruijn. The reversed complement of a de Bruijn sequence S is also a de Bruijn sequence and identical to the complement reversed de Bruijn sequence of S. □

The set of connected de Bruijn sequence

Define the set of connected de Bruijn sequence as the set containing a de Bruijn sequence, the reverse sequence, the complemented sequence and the reverse complement of it. So any de Bruijn sequence is a part of such a set, that can contain up to four unique de Bruijn sequences. The possible number of unique sequences in such a set is 1,2 and 4.

Lemma 2. *It is impossible for a set of connected de Bruijn sequence to contain exactly 3 unique de Bruijn sequences.*

Proof. Prove this by showing that it is impossible to construct a set of exactly 3 unique de Bruijn sequences. For each part S is refer to a reference de Bruijn sequences, $R(S)$ is the reverse sequence of S, $C(S)$ is the compliment sequence and $RC(S) = CR(S)$ is the reverse complement(or complemented reverse) of the sequence.

Assume that S, $R(S)$ and $C(S)$ are all unique, and $RC(S) = CR(S)$ is equal to S,$R(S)$ or $C(S)$. If $C(S) = CR(S)$ then, $C(S) = C(CR(S) = R(S)$ which is a contradiction. If $S = RC(S)$ then, $R(S) = R(RC(S) = C(S)$ which is a contradiction. If $R(S) = RC(S)$ then, $R(R(S)) = S = C(S)$ which is a contradiction. So S,$R(S)$ and $C(S)$can not be unique while $RC(S)$ is not.

Assume that S, $R(S)$ and $RC(S)$ are all unique, and $C(S)$ is equal to S,$R(S)$ or $RC(S)$. If $C(S) = CR(S)$ then, $C(S) = C(CR(S) = R(S)$ which is a contradiction. If $C(S) = S$ then, $RC(S) = CR(S) = R(S)$ which is a contradiction. If $C(S) = CR(S)$ then, $S = R(S)$ which is a contradiction. So S,$R(S)$ and $RC(S)$can not be unique while $RC(S)$ is not.

Assume that S, $RC(S)$ and $C(S)$ are all unique, and $R(S)$ is equal to S,$RC(S)$ or $C(S)$. If $R(S) = CR(S)$ then, $S = C(S)$ which is a contradiction. If $R(S) = S$ then, $CR(S) = C(S)$ which is a contradiction. If $R(S) = C(S)$ then, $CR(S) = S$ which is a contradiction. So S,$R(S)$ and $C(S)$can not be unique while $RC(S)$ is not.
So all possible the combination of only 3 unique set of de Bruijn sequences all leads to contradictions, so it is impossible to have any set of exactly 3 unique de Bruijn sequences. □

A de Bruijn sequences in a set of connected de Bruijn sequences will have the same relative security as the other sequence in the set, as it is easy to find any of them from any of the other sequences.

3.8 Problems

The field of *NLFSR* have relatively many unsolved problem, little is known about any possible alternative representation of *NLFSR*. Below is a list of the most essential problems that have yet to be solved.

Finding the period of any *NLFSR*

One of the main problem of this particular field of research, and one of the hardest. Given a general method for this would vastly improve the possibility of the field.

Determine if a *NLFSR* generates a de Bruijn sequence

These is a sub problem of the first problem and the most important usability of these problem.

Bound on *NLFSR* and classes of *NLFSR*

Finding bounds on complexity or period for *NLFSR* classes. That gives sequences with a lower bound on the period and linear complexity. Symmetric shift register and de Bruijn is examples of classes where this is found.

Find a ways to efficiently generate de Bruijn sequences

A relatively hard problem, there exist algorithm to find this from de Bruijn sequences of lower orders. But the efficiency scales quite badly for large orders, either in time or memory requirement. Some algorithm to find special de Bruijn sequences for all order higher than 2, as the "Prefer" one and "Prefer same" one. But these sequences have low security as they are easy to find.

There is also algorithm for recursively find new de Bruijn sequence from others of lower orders. Lempels inverse D-homomorphism can be applied to a de Bruijn sequence in B_n to produce two cycles D_1 and D_2 of length 2^n in B_{n+1}. Any changing of successors of conjugate pair that are on different cycles (D_1 and D_2) produces a cycle of length 2^n in B_{n+1}, which is a de Bruijn sequence of order $n + 1$.

In the next section we will give a list of algorithms and methods to generate de Bruijn sequences.

4 Algorithms and methods to find de Bruijn sequence

Most of these algorithms are explored more fully in a survey by Fredricksen [9].

4.1 Make de Bruijn sequences form *LFSRs*

From an M-sequence

Let $f(x_0, x_1, ..., x_{n-1}) = x_0 \oplus g(x_1, .., x_{n-1})$ generate an M-sequence of order n, then a de Bruijn sequence can be created by joining the two cycles generated by f. As changing one of the values of the truth table of g will change the successor of two states are swapped, this can be used to join to cycles. If the two states are on different cycles, the cycles will be joined. But if they are on the same cycle, the cycle will be split into two.

A maximum-length *LFSR* have two cycles the all zero cycle of period one and the main cycle with all the other states, that has a period of $2^n - 1$. Then by simply changing the first entry in the truth table $(0, 0, 0, ..., 0) \Rightarrow 0$ to $(0, 0, 0, ..., 0) \Rightarrow 1$, the two cycles are joined into a single cycle of length 2^n. This is a de Bruijn sequence that can have maximum linear complexity of $2^n - 2$. But it has a relatively low security, as it is easy to find the original M-sequence by simply using the Berlekamp-Massey algorithm.

From *a FSR* by using cycle joining

Given a non-singular *FSR* with c cycles, it is possible to make a de Bruijn sequence by iteratively join one and one cycle. So after $(c - 1)$ such operations the result is a de Bruijn sequence. E.g. start with the PCR which has $Z(n)$ cycles, can be joined into a de Bruijn sequence in $Z(n) - 1$ operations. This gives a truth table of weight $Z(n) - 1$, so the truth table can be presented by $Z(n) - 1$ integers in the range form 0 until 2^n. Then $n \cdot (Z(n) - 1) = n \cdot Z(n) - n$ bits are needed to represent the integers, this can be further optimized as there are some values that have to be 1. E.g. there is only one way to join some cycles, as the all zero cycle.

4.2 From some *LFSR*s with reducible characteristic polynomials

C. Li, X. Zeng, C. Li, T. Helleseth and M. Li [16] gives a method to find a class of de Bruijn sequences from a reducible polynomial

$$q(x) = p_0(x) \cdot p_1(x) \cdot p_2(x) \cdot ... \cdot p_k(x),$$

$p_i(x)$, $0 \leq i \leq k$ are all primitive polynomials. The degree of $p_j(x) > p_{j-1}(x)$, $1 \leq j \leq k$ and $p_0 = (1 + x)$. n is the degree of $q(x)$ and d_k is the degree of p_k. This process have time complexity $O(2^{n-d_k} \cdot k \cdot n)$ and memory complexity $O(2^k \cdot k \cdot n)$. If $n \geq 8$ it is possible to have time complexity of $O(n^{log_2^2 n})$.

This is done determining the cycle structure and adjacency graph.

A class of de Bruijn sequences

C. Li, X. Zeng, C. Li, T. Helleseth [15] gives a method to find a class of de Bruijn sequences from the polynomial $q(x) = (1 + x^3) \cdot p(x)$, $p(x)$ is a primitive polynomial of degree $n > 2$. Which is done by determining the cycle structure and adjacency graph of $q(x)$.

4.3 Iterative construction of the "Prefer one" de Bruijn sequence

The "Prefer one" de Bruijn sequence is the sequence constructed by starting with the all zero state, and for each current state $S_i = (s_i, s_{i+1}, ..., s_{i+n-1})$ check if the state $S_1 = (s_{i+1}, s_{i+2}, ..., s_{i+n-1}, 1)$ have already occurred in the sequence. If it have not, then the next state is S_1. And if it have, the next state is then $(s_{i+1}, s_{i+2}, ..., s_{i+n-1}, 0)$.

Simple algorithm

1. Start with the all zero state $(0, 0, 0, ..., 0)$.
2. The current state is $S_i = (s_i, s_{i+1}, ..., s_{i+n})$, then if the state $S_{i+1} = (s_{i+1}, ..., s_{i+n}, 1)$ has not previously appeared in the sequence then the S_{i+1} is the next state and repeat part 2. Otherwise go to part 3.
3. The current state is $S_i = (s_i, s_{i+1}, ..., s_{i+n})$, then if the state $S_{i+1} = (s_{i+1}, ..., s_{i+n}, 0)$ has not previously appeared in the sequence then the S_{i+1} is the next state and go to part 2. Otherwise the algorithm is finish.

This algorithm generate a de Bruijn sequence in $O(2^n)$ operations, but require $O(2^n)$ memory, so it is not usable for generating sequences of orders that requires more that the available memory.

Advanced algorithm

This is an advanced algorithm for finding the "Prefer one" sequence.

1. From $\beta_i = (b_1, b_2, ..., b_n)$ produces $\beta_{i+1} = (b_2, ..., b_{n+1})$. If $(b_1, b_2, ..., b_n) = (b_1, 1, 1, ..., 1)$, the next n-tuple is $(1, 1 - ..., 1, \bar{b}_1)$. Otherwise go to step 2.
2. $\beta_i^* = (b_2, ..., b_n, 1)$. Go to step 3
3. Find the largest state $m_i = (b_s, ..., b_n, 1, b_2, ..., b_{s-1})$ of all the cyclic shifts of β_i^*. Go to step 4.
4. If $b_2 = b_3 = ... = b_{s-1} = 0$, then $\beta_{i+1} = (b_2, ..., b_n, \bar{b}_1)$. Otherwise $\beta_{i+1} = (b_2, ..., b_n, b_1)$. Go to step 1.

This algorithm only requires $(3 * n)$ bits to store the current value for β_i^*, β_i and the current larges value of cyclic shift of β_i^*. The time complexity to generate a bit is $O(n)$, as the operation to find m_i can take n operations. Then the complexity to generate a whole de Bruijn of order n is $O(n \cdot 2^n)$. Also the simple algorithm have to start at the all zero state, but this can start at any state.

Finding new sequences form the "Prefer one" by using cross-join pair

Because the structure of the "Prefer one" sequences is known, it is possible to construct 2^{2n-5} de Bruijn sequences by finding different cross-join pair. The 3 following method finds independent sets of cross-join pairs.

1: The number $a_i, b_i = 2^i, 2^{n-2-i}$ determines nonintersecting cross-join pairs for $0 \leq i \leq [n - \frac{3}{2}]$.

2: The numbers $a_i, b_i = 2^{n-1} - 1 - 2^i, 2^{n-1} - 1 - 2^{n-2-i}$ determines nonintersecting cross-join pair for $0 \leq i \leq [n - \frac{3}{2}]$.

3: The numbers $a_i, b_i = 2^i - 1, \Sigma_{j=1}^{i} 2n - 1 - j$ determine nonintersecting cross-join pair for $1 \leq i \leq n - 2$.

These pairs can then be used to generate a de Bruijn sequence that differ form the "Prefer one"'s truth table, by two times the number of cross-join pairs

4.4 Iterative construction of the "Prefer same" de Bruijn sequence

The "prefer same" is a de Bruijn sequence constructed in much the same way as the "Prefer one", but instead of preferring that the next bit of the sequences is a 1, the preferred next bit is the same as the current last bit.

1. Start with n ones followed by n zeroes, on the form:

$$x_1 x_2 ... x_{n+1} x_{n+2} ... x_{2n} = 11...00...0.$$

2. Next bit of the sequences x_k,$k \geq (2 \cdot n)$ is equal to the previous bit $x_{k-1} = i$ if the two following conditions are fulfilled. If they are, then repeat step 2. Otherwise go to step 3.

    - The n-tuple $x_{k-n+2} ... (x_k = i)(x_{k+1} = i)$ has not previously appeared in the sequence.
    - If placing $x_{k+1} = i$ makes a string of i's of length t, then we have not already written 2^{n-2-t} strings of i's of length t in the sequence.

3. Set $x_{k+1} = \bar{i}$ if this does not violate the two conditions of step 2, and got to step 2. If not, then the sequence is complete and the algorithm terminate.

As the simple algorithm for finding the "Prefer one" de Bruijn sequence, this algorithm uses $O(2^n)$ bit of memory. There is no known advanced algorithms that lowers the requires amount of memory needed to generate this sequence.

4.5 Generating de Bruijn sequences by recursively using the inverse D-homomorphism

Lempels D-homomorphism [2] is a transformation from any cycle B of length m in B_{n+1} to two cycles C and its complement $\overline{C}$, both of length m in B_n. If $C = (c_0, c_1, c_2, ..., c_{m-1})$ then the D-homomorphism:

$$D(C) = (c_0 \oplus c_1, c_1 \oplus c_2, c_2 \oplus c_3, ..., c_{m-1} \oplus c_0) = B$$

$$D(\overline{C}) = (\overline{c}_0 \oplus \overline{c}_1, \overline{c}_1 \oplus \overline{c}_2, \overline{c}_2 \oplus \overline{c}_3, ..., \overline{c}_{m-1} \oplus \overline{c}_0) = B$$

Its inverse is more interesting, as it makes it possible to find de Bruijn sequences of higher orders recursively.
Lempels inverse D-homomorphism is a transformation from a cycle C of length m in B_n to two cycles $D_0^{-1}(C)$ and $D_1^{-1}(C)$ of length m in B_{n+1}. Such that $D_0^{-1}(C)$ and $D_1^{-1}(C)$ does not share any states in B_{n+1}, since they do not share any states in B_{n+1} they can be generated by a feedback function f. If $C = (c_0, c_1, c_2, ..., c_{m-1})$ then:

$$D_0^{-1}(C) = (0, c_0, c_1 \oplus c_0, c_2 \oplus (c_1 \oplus c_0), ..., c_{m-2} \oplus (c_{m-3} \oplus ......))$$

$$D_1^{-1}(C) = (1, c_0 \oplus 1, c_1 \oplus (c_0 \oplus 1), c_2 \oplus (c_1 \oplus (c_0 \oplus 1)), ..., c_{m-2} \oplus (c_{m-3} \oplus ......))$$

Then by swap of any pair of complement state, such that one is in $D_0^{-1}(C)$ and another in $D_1^{-1}(C)$. Produces a cycle of length $2 * m$ in B_{n+1}. As the sequences $D_0^{-1}(C)$ and $D_1^{-1}(C)$ are obviously each others complement complement, if a state $s \in D_1^{-1}(C)$, then $\overline{s} \in D_2^{-1}(C)$ and vice versa

$$s \in D_1^{-1}(C) \Longleftrightarrow s \notin D_2^{-1}(C)$$

The two alternating states $(0, 1, 0, 1, 0, 1, ...)$ and $(1, 0, 1, 0, 1, 0, ...)$ both of length $n + 1$ have the unique properties that they are both each others complement, predecessor and successor in B_n. I.e. changing the successors of the states $(x, 1, 0, 1, 0, 1, ...)$ or $(x, 0, 1, 0, 1, 0, ...)$ will create a de Bruijn sequence in B_{n+1}, if the original sequence C is a de Bruijn sequence in B_n.

E.g. given a de Bruijn sequence $S = (0, 0, 1, 1)$ of order 2. Then $D_0^{-1}(S) = (0, 0, 0, 1)$ and $D_1^{-1}(C) = (1, 1, 1, 0)$, both cycles of length 4 and order 3. Then by changing the truth table for the entry $(x, 1, 0)$ gives the sequence $(0, 0, 0, 1, 0, 1, 1, 1)$.

The D-homomorphism on the Boolean function

Lempel [2] also introduces an operation to apply the D-homomorphism directly on the recursive feedback function. For the original function $\phi(x_0, x_1, ..., x_{n-1})$ and let the inverse D-homomorphism be $f(x_0, x_1, ..., x_n)$. The relation between them is as follows:

$$f(x_0, x_1, ..., x_n) = x_n \oplus \phi((x_0 \oplus x_1), (x_1 \oplus x_2), ..., (x_{n-2} \oplus x_{n-1}))$$

This makes it possible to find the inverse D-homomorphism of a function directly, and the complexity of this operation depends on the number of terms of the original function ϕ, and the number of variables for each term. The resulting function f will have Σv_i terms, where v_i is the number of variables a term of ϕ. So before any shortening of f, the number of terms is $term(f) \geq 2 \cdot term(\phi)$. The number of terms after a shortening is bounded by $2^n \geq term(f) \geq 0$.

If ϕ is a de Bruijn sequence, the two cycles generated by f can then be joined by adding the term $\overline{x}_1 x_2 \overline{x}_3, ...$ or $x_1 \overline{x}_2 x_3, ...$ that corresponds to one of the two alternating states to f. The problem of finding a feedback function for this on ANF form is generally hard. But since this function transformation can be applied to function that is not on ANF form, so this is not really a problem. But it does increase the number of possible unique terms from 2^n to $2^{2 \cdot n}$.

So an algorithm for finding a function that generates a de Bruijn sequence of order n from a function that generates a de Bruijn sequence of order m by the inverse D-homomorphism. The time and memory needed to do this, will depend on the number of terms for each recursive step from an order k to order $k + 1$. As there exist at the time no way to accurately calculate this, this mean the cost in time and memory is unpredictable and the function may then not terminate in a reliable time interval.

4.6 Games generalization of the D-homomorphism

Games [10] gives a method to construct a de Bruijn sequence S of order $n + 1$, from two de Bruijn sequences S_1 and S_2 of order n. That relates the linear complexity of S_1 and $S_2$to the linear complexity of S. If both S_1 and S_2 have a linear complexity of $2^n - 1$, then S also have a linear complexity of $2^{n+1} - 1$.

Definition 9. The *image of a sequence $S = (s_0, s_1, ..., s_{n-1})$ of order n is denoted $Im(S)$. That is the set of states of length n that occur in S. The shift operation is:*

$E^i(S) = (s_i, s_{i+1}, .., s_{i-2}, s_{i-1})$

The Method is generalized here as a theorem:

Theorem 7. *If r, s are two de Bruijn sequences of order n, such that $Im(D_0^{-1}(r)) = Im(D_0^{-1}(s))$ and hence $Im(D_1^{-1}(r)) = Im(D_1^{-1}(s))$.*

For any i, j such that:

$E^i(D_0^{-1}(r)) = (u_0, u_1, ..., u_{2^n-n-2}, a_0, a_1, ..., a_n)$ *and*

$E^j(D_1^{-1}(r)) = (v_0, v_1, ..., v_{2^n-n-2}, \overline{a}_0, a_1, ..., a_n)$

Then $E^i(D_0^{-1}(r)) || E^j(D_0^{-1}(r)) =$
$(u_0, u_1, ..., u_{2^n-n-2}, a_0, a_1, ..., a_n, v_0, v_1, ..., v_{2^n-n-2}, \overline{a}_0, a_1, ..., a_n)$
is a de Bruijn sequence of order n+1.

This holds for any conjugate states $(a_0, a_1, ..., a_n)$ and $(\overline{a}_0, a_1, ..., a_n)$ that are on different image sets. And if $r = s$ this is just Lempels inverse D-homomorphism, and a joining of the cycles by interchanging the successor of conjugate states on $D_0^{-1}(C)$ and $D_1^{-1}(C)$. So this method can be seen as a generalization of this.

Games [10] also looks at reverse de Bruijn that have the same type-0 images, that is $Im(D_0^{-1}(S)) = Im(D_0^{-1}(R(S)))$. These de Bruijn sequences are called *self* D_0^{-1}*image reversing sequences,* it is shown that the order of the these have to be even. For any *self* D_0^{-1}*image reversing sequences* r of order n, $s = D(D_0^{-1}(r))[a\,, b;\, c\,, d]$ and $[a\,, b;\, c\,, d]$ is a *cross-join pair* of $D_0^{-1}(r)$. Then s is also a *self* D_0^{-1}*image reversing sequences* of order n, that is not equal to r.

4.7 Annexsteins algorithm

Annexsteins [1] gives an efficient implementation on recursively generating de Bruijn sequences of order n from any de Bruijn sequence of order $n > m \geq 2$ by using the inverse of Lempels D-homomorphism. This work directly on the de Bruijn sequence and sub parts of it, taking sub sequences, complementing some and concatenate the results recursively for each order.

This requires $O(2^n)$ operations to generate a de Bruijn sequence of length 2^n, but it is bound by a memory constrain as sub sequences of the de Bruijn sequence for an order will have to be stored in the memory. So the memory complexity is $O(2^n)$, and therefore is not applicable to generating de Bruijn sequences of high orders.

For each recursive step a binary decision is made between using one of the two alternating sequence of length $(k-1)$, for a total number of 2^{n-m} combinations. Which correspond to the 2^{n-m} unique de Bruijn of order that can be generated this way from a starting de Bruijn sequence. Each unique sequence can be identified by a $(n-m)$ bit string that determine the decision for each recursive step. E.g. the second bit indicate that the alternating sequence used at the recursive step number 2 is 0101... if it is 0 or 1010... if it is 1.

4.8 Generating de Bruijn sequences by non-recursively using the inverse D-homomorphism

Chang, Park, Kim and Song [4] present an algorithm to generate de Bruijn sequence of order n from a de Bruijn feedback function of order k. The number of *exclusive or* operation needed to generate one bit is approximately $k(2^{W(n-k)} - 1)$, and $W(r)$ is the weight of the binary representation of r. So if $W(n-k) = 1$, the efficiency depends only on k.

Lemma 3. *If m is a nonnegative power of 2:*
$D^m(x_0,x_1, ..., x_m) = x_0 \oplus x_m$

This gives an expression of non-recursively using the D-homomorphism, if the number of state variables is one more than the number of time the D-homomorphism transformation used.

Lemma 4. *If m is a nonnegative power of 2 and $n > m$:*
$D^m(x_0, x_1, ..., x_{n-1}) = (x_0 \oplus x_m, x_1 \oplus x_{m+1}, ..., x_{n-m} \oplus x_n)$

The following algorithm finds $\Delta_i^n(x)$, which is used in the theorem 9:

Algorithm to find $\Delta_i^n(x)$

Define $\phi = (0, 0, 0, ...)$, $\delta = (1, 0, 1, 0, ...)$, $\alpha = (0, 1, 0, 1, ...)$ and $\tau = (1, 1, 1, 1, ...)$. Let $\phi_n, \delta_n, \alpha_n, \tau_n$ be the sequence consisting of the first n bits of $\phi, \delta, \alpha, \tau$.

Given inputs n,i and x:

```
1. If $x = \delta_n$, then $\Delta_i^n(x) := 1$ and terminate. If $x = \alpha_n$, or $x = \tau_n$, or
   $x = \phi_n$, then $\Delta_i^n(x) := 0$ and terminate. Else move to step 2.
2. $d ;= n - i + 1$ go to step 3.
3. if $D^d(x) \neq \phi_{n-d}$, then $\Delta_i^n(x) := 0$ and terminate. Else go to step 4.
4. $k := 1$,$m := d - 1$
5. while $k < m$ do
6. t := $\lfloor \frac{k+m}{2} \rfloor$
7. if $D^t(x) = \delta_{n-t}$, then $\Delta_i^n(x) := 1$ and terminate.
8. if $D^t(x) = \alpha_{n-t}$, then $\Delta_i^n(x) := 1$ and terminate.
9. if $D^t(x) = \tau_{n-t}$, then $t := t - 1$ go to step 7.
10. if $D^t(x) = \phi_{n-t}$, then $m := t - 1$ else $k ;= t$.
11. Terminate.
```

This algorithm may terminate after just a few step and [4] show that it is probable that it will not run longer than step 3.

Definition 10. Let $R : B^n \longrightarrow B^{n-1}$be a projection function:
$R((x_0, x_1, ..., x_{n-1})) = (x_0, x_1, ..., x_{n-2})$
Let $S : B^n \longrightarrow B^n$ be the shift left function
$S((x_0, x_1, ..., x_{n-1})) = (x_1, x_2, ..., x_{n-1}, 0)$
Let $P_i : B^n \longrightarrow B$ be the bit selection function
$P_i((x_0, x_2, ..., x_{n-1}) = x_i,\ 0 \leq 1 \leq n-1$

Theorem 8. *For an integer $i \in \{1, 2, ..., n\}$,we have*
$P_{n-i} \cdot D^i \cdot S(x^{(n)}) = x_n^{(n)} \oplus x_{n-1}^{(n-1)} \oplus ... \oplus x_{n-i+1}^{(n-i+1)}$

This theorem is used in the next theorem.

Theorem 9. *Let h_k be a k-DBF. Then for $x^{(n)} \in B^n$*
$h_n(x^{(n)}) = P_k \cdot D^{n-k} \cdot S(x^{(n)}) \oplus h_k(D^{n-k}(x^{(n)})) \oplus \Delta_{k-1}^{n-1}(R \cdot S(x^{(n)}))$
is an n-DBF.

Using this, it is possible to generate an n order de Bruijn sequence from an *k-DBF*. Without needing to find the Boolean function. The only downsize that this *k-DBS* is equal to the one that is the product of using the recursive reverse *D-homomorphism* and using the same alternating state for all the recursive steps. Meaning that given the original *n-DBF* there is is only two possible *k-DBS* that can be generated by this method alone. Making this operation highly predictable and the final de Bruijn sequence being unsecure.

In the next section a method to use the inverse D-homomorphism to generate a de Bruijn sequence of order $(n+1)$ from a de Bruijn sequence of n with only needing $O(log_2 n)$ memory and $O(log_2^2 n)$ operations. Also is a hardware implementation to recursively use this to produces 2^{n-m} unique de Bruijn sequences of order n for any input de Bruijn sequence of order m.

5 Hardware implementation to generate de Bruijn recursively

This section present a method to use the inverse D-homomorphism on a de Bruijn sequence of order n, to generate a de Bruijn sequence of order $(n+1)$ without storing the state. While only using $O(log_2 n)$ bits of memory and $O(log_2^2 n)$ logic operations. Later in the section a hardware implementation of this is shown, that can be connected in series to a generating de Bruijn sequences of order n from a de Bruijn sequence of order m.

The main focus here is on simply generating de Bruijn sequence of relatively high orders. To clarify this order 150 and 300 are here low orders, and order higher than 10000 are high. Many of the known algorithms and methods to generate de Bruijn sequences have relatively bad scaling in either the time or memory requirement to generate the sequences.

5.1 Method to implement the inverse D-homomorphism without storing the state

The reverse D-homomorphism of a de Bruijn sequence $S = (s_0, s_1, ..., s_{2^n-1})$ of order n produces two new primitive cycles $D_0^{-1}(S)$ and $D_1^{-1}(S)$, both of length 2^n:

$$D_0^{-1}(S) = (0, s_0 \oplus 0, s_1 \oplus (s_0 \oplus 0), ..., s_{2^n-1} \oplus (...)) = (a_0, a_1, ..., a_{2^n-1})$$

$$D_1^{-1}(S) = (1, s_0 \oplus 1, s_1 \oplus (s_0 \oplus 1), ..., s_{2^n-1} \oplus (...)) = (b_0, b_1, ..., b_{2^n-1})$$

Where $a_k = \bar{b}_k$, $0 \leq k \leq 2^n - 1$ and it is possible to change from ith entry of $D_0^{-1}(S)$ to the ith entry of $D_1^{-1}(S)$ by complementing the ith bit of the sequence. Notice that the input to $D_0^{-1}(S)$ and $D_1^{-1}(S)$ for bit a_{i+1}/b_{i+1} is the bit s_i and a_i/b_i. The states 101... and 010... are the alternating states of length $(n+1)$ of $D_0^{-1}(S)$ and $D_1^{-1}(S)$, and they have the property that they are not only the complement of each others, but also each others possible predecessor and successor state. Since every state has two possible predecessor and successor and 101... and 010... are on different cycles, it is possible to deduce there predecessor and successor. Below the deduced sub sequence for odd and even order is shown.

$$\rightarrow 110* \rightarrow 101* \rightarrow 0*11 \rightarrow$$

$$\rightarrow 001* \rightarrow 010* \rightarrow 1*00 \rightarrow$$

$$\rightarrow 1101* \rightarrow 1010* \rightarrow 01*00 \rightarrow$$

$$\rightarrow 0010* \rightarrow 0101* \rightarrow 10*11 \rightarrow$$

$*$ can be any of the two alternating sequences 01... and 10... of even length, that satisfied the condition that the bit before the sequence is different from the first bit of the sequence. And the last bit of the sequence is different from the first bit after the sequence, if there is any. Let x denote either 1 or 0.

The cycles $D_0^{-1}(S)$ and $D_1^{-1}(S)$ can be joined by change the successor of the two states on the form $x01*$ or $x10*$ for even length $(n+1)$. Or the two states on the form $x010*$ or $x101*$ for odd length $(n+1)$. But this also changes the phase of the output sequence in comparison to the input sequence S. E.g. if the successor of the states on the form $x01*$ is changed, starting in the cycle containing the sub sequence $\rightarrow 110* \rightarrow 101* \rightarrow 0*11 \rightarrow$ gives the following two parts of the joined cycle.

$$\rightarrow 110* \rightarrow 101* \rightarrow 010* \rightarrow 1*00 \rightarrow$$

$$\rightarrow 001* \rightarrow 0*11 \rightarrow$$

So instead of going to the state $0*11$, it goes to $010*$. If look at the deduced sub sequences of $D_0^{-1}(S)$ and $D_1^{-1}(S)$, $010*$ is one state earlier than $0*11$. So the input S have to be shifted minus one entries, i.e. to find the next bits of the joined sequence, s_{i+1} will have to be used instead of s_i.

This continues until the state $001*$ is reached, its new successor is $0*11$, instead of $010*$. Here $0*11$ is one state later than $010*$. So the input S have to be shifted plus one entries, back to the original. I.e. to find the next bits of the joined sequence, s_i will have to be used instead of s_{i+1}.
Let denote this shift as "phase" of the input.

Definition 11. Phase is a variable for the input to the function denoted p, and the input to the function is bit $s_{i+p \, mod(2^n-1)}$ of S at time $(i+1)$.

Theorem 10. *The phase shift is determined by the two first bit of the selected alternating state on the form $x01*$ or $x10*$. If they are the same the phase have to be shifted by $+1$ and if they are different it will have to be changed -1. Phase can only take on the values 0,-1,$+1$.*

Proof. Prove this by showing that for n even or odd, the alternating states both follows this rule.

n even:
x01$*$: this gives the following parts of the DBS.

$\to 110* \to 101* \to 010* \to 1*00 \to$
$\to 001* \to 0*11 \to$
$101*$ new successor $010*$ is one state earlier than its original successor state $0*11$. So the phase is -1, which is in accordance with the stated hypothesis.
$001*$ new successor $0*11$ is one state later than its original successor state $010*$. So the phase is $+1$, which is in accordance with the stated hypothesis.

x10$*$: this gives the two following sub parts of the DBS.

$\to 110* \to 1*00 \to$
$\to 001* \to 010* \to 101* \to 0*11 \to$
$010*$ new successor $101*$ is one state earlier than its original successor state $1*00$. So the phase is -1, which is in accordance with the stated hypothesis.
$110*$ new successor $1*00$ is one state later than its original successor state $101*$. So the phase is $+1$, which is in accordance with the stated hypothesis.

n odd:
x010$*$: this gives the following parts of the DBS.

$\to 1101* \to 1010* \to 0101* \to 10*11 \to$
$\to 0010* \to 01*00 \to$
$0010*$ new successor $01*00$ is one state earlier than its original successor $010*1$. So the phase is $+1$, which is in accordance with the stated hypothesis.
$1010*$ new successor $0101*$ is one state later than its original successor $01*00$. So the phase is -1, which is in accordance with the stated hypothesis.

x101$*$: this gives the following parts of the DBS.

$\to 1101* \to 10*11 \to$
$\to 0010* \to 0101* \to 1010* \to 01*00 \to$
$0101*$ new successor $1010*$ is one state earlier than its original successor $10*11$. So the phase is -1, which is in accordance with the stated hypothesis.
$1101*$ new successor $10*11$ is one state later than its original successor $1010*$. So the phase is $+1$, which is in accordance with the stated hypothesis. □

So it is possible to determine the phase after any alternating sequence on the form $x01*$ or $x10*$ and this makes it possible to join the cycle $D_0^{-1}(S)$ and $D_1^{-1}(S)$ into two unique de Bruijn sequences of order $(n+1)$.

Theorem 11. *This method can be used to generate 2^{n-m} unique de Bruijn sequences of order n for all de Bruijn sequences of order m.*

Proof. [2] proves that the D-homomorphism has a one-to-one correspondence between S and the pair $D_0^{-1}(S)$, $D_1^{-1}(S)$ if S has an even weight. De Bruijn sequence have even weight, so $D_0^{-1}(S)$, $D_1^{-1}(S)$ is unique for every de Bruijn sequence S. Let S_{+1} be the resulting de Bruijn sequence from joining $D_0^{-1}(S)$, $D_1^{-1}(S)$ for some de Bruijn sequence of order n S, by changing the successors of the state $x01*$ or $x10*$.

Assume that $S_{+1}$is created by joining $D_0^{-1}(S)$, $D_1^{-1}(S)$ by changing the successor of $x01*$, and the feedback function for S_{+1} is $f(x_0, x_1, ..., x_{n-1}) = x_0 \oplus g(x_1, .., x_{n-1})$. Then the only difference between the feedback function that generates $D_0^{-1}(S)$, $D_1^{-1}(S)$ and the one generating S_{+1}, is the entry of $g(0, ..., 1)$. Then is there then any $S' \neq S$ such that joining $D_0^{-1}(S')$, $D_1^{-1}(S')$ by changing the successor of the state $x01*$ or $x10*$, equals S_{+1}.

The changing the successor of $x01*$ of S_{+1} gives $D_0^{-1}(S)$, $D_1^{-1}(S)$, and since there is a one-to-one correspondence between S and the pair $D_0^{-1}(S)$, $D_1^{-1}(S)$. S_{+1} cannot be constructed from S' by joining $D_0^{-1}(S')$, $D_1^{-1}(S')$ changing the successor of $x0...1$.

The changing the successor of $x10*$ of S_{+1} gives 2 cycles, one of length 2^n-2 and 2. As S_{+1} was constructed by changing the successor of $x0*1$, which result in that the successor to the state $101*$ is $010*$. And the changing of the successor of $01*0 = 010*$ to $101*$ creates the cycle of length 2. Which means that it is not possible to construct S_{+1} from a $S' \neq S$ by changing the successors of the state $x01*$ or $x10*$ to join $D_0^{-1}(S')$, $D_1^{-1}(S')$. So every possible de Bruijn sequence of order n can be used to create two unique de Bruijn sequence of order $n+1$. □

By only looking at the two first bit of the selected form and count the number of alternating output bits up to $(n-1)$. In the following sections a hardware implementation of this is shown, that have a component complexity of $O(n \cdot log_2 n)$ and a time complexity to produce a bit of $O(log_2^2 n)$.

5.2 Introduction to the hardware implementation

When any computational task is to be performed, there is generally two ways to do this. The first way is to solve it by writing a software program and run this program on a computer to get the result. The other way to is to create a special piece of hardware that will solve the problem. The software way is general solution as there will be no additional cost in physical resources and it can be performed on any computer.

The hardware way makes it possible to divide tasks into multiple separately independent operations. Which can then be run in parallel to solve the problem faster. Will in this part look at a hardware implantation to generate long de Bruijn sequences from a shorter de Bruijn sequence using Lempels inverse D-homomorphism.

All modern digital electronics is primarily created by using logic gates. A logic gate take a number of binary input and gives binary output in accordance with the truth table of the gate. The most simple are AND, NOT, OR and XOR logic gates. In addition there is the complement of OR, NOR. By using these as building blocks, its possible to construct complex logical systems. To measure the efficiency of system, one have to look at the cost in resources to build the system and the cost to operate it.

When any hardware generator is implemented, the two main resources are time and component. The most important cost in time, is the time to solve a task and to implement it into a system that generate the sequence. The time needed to generate one sub part of the main task.

For components, the number of transistor is a good measurement of efficiency, another one is the number of logic gates and memory cells. Here the sum number of logic gates and memory is used to indicate the complexity of the system. As logic gates and memory can be constructed from primarily transistor and memory cells, this is a relatively good measurement of the complexity of the system.

Use the notation $(n-m)$-incrementer to prefer to the hardware system. The task of this system is to take an input stream, that is a de Bruijn sequence of order m. And output a de Bruijn of order n, without there being any delay in the difference between the two. I.e. every time single bit of the input stream is sent to the system, a single bit of output is delivered from the system. Below in figure 9 is basic overview of the $(n-m)$-incrementer.

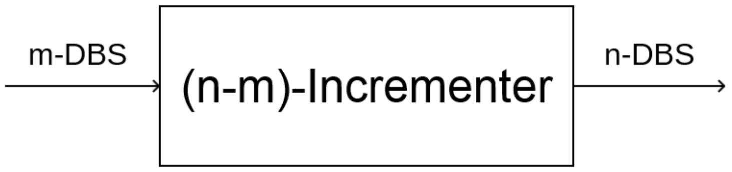


Figure 9: An overview of the $(n-m)$-incrementer

The $(n-m)$-incrementer consist of $(n-m)$ independent sub system called incrementers. These are independent, as they can operate at the same time without any of the incrementers having to wait for an input from another for every clock cycle. The clock frequency is limited by the slowest incrementer, as all the incrementers have to complete there task before the system can clock. Also incrementers can use a shared memory cells, as long as only one incrementer sets the value of the cell.

An incrementer takes as an input a de Bruijn sequence of order k and output a de Bruijn sequence of order $(k+1)$. In figure 10 below, it is shown an illustration of how the incrementers are used as building blocks of the $(n-m)$-incrementer.

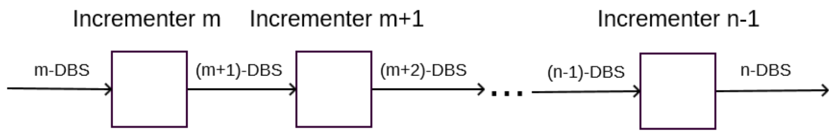


Figure 10: An overview of the $(n-m)$-incrementer

5.3 Components

This hardware implementation uses AND, XOR, NOT gates and D flip-flops. It can be further optimized by using some of the complement gates, but for simplicity only the simple implementation is used. All of these component can be made from transistors in several different ways, that gives different properties in terms of the number of transistors used and propagation delay. The most important properties of the logic gates in an overhead view is:

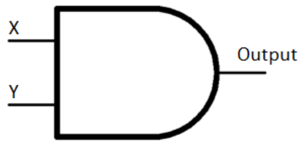
Definition 12. *Propagation delay* is the maximum time it takes for the output of a logic gate to change, when there is a change in the input to the logic gate. This limits the speed of which the logic block can operate without there being a logical error.
Fan-out is the maximum number of parallel logic gate that the output of a logic gate can give a signal to, without there being a logical error.

The Propagation delay of a logic block is the longest time for the blocks output to change into its final state, when any input to the block changes. This depend on the "longest path" of the block, which is the longest path of components an input signal has to travel to an output. So the total propagation delay is the sum of the propagation delay of the components on the "longest path". Of course this is only applicable to a logic block that stabilizes for all inputs. I.e. that all output will eventually reach a constant truth value, for all input.

While there may be a more efficient ways to construct the logic blocks, by using these few components the design does not get to complex to understand. No electrical analysis on the power levels, noise cancellation, boosting of signal and other details is performed. As this is only meant as a basic look at how generally efficient this method of generating de Bruijn sequence can be implemented in hardware circuit.

AND gate

An AND gate takes two binary input and gives a 1 if both inputs are 1, and 0 otherwise. This is illustrated in the truth table below to the right, on the left is the symbol used in block schematic to indicate an AND gate. But in all figures in this thesis use the symbol $\otimes$.



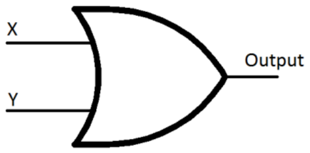
X	Y	Output
0	0	0
0	1	0
1	0	0
1	1	1

Figure 11: Symbol and truth table for an AND gate

This logic gate can be used to check if two Boolean values are both 1.

OR gate

An OR gate takes two binary input and gives a 1 if any of the inputs are 1, and 0 otherwise. This is illustrated in the truth table below to the right, on the left is the symbol used in schematic to indicate an OR gate. But in all figures in this thesis use the symbol $\bigcirc$.



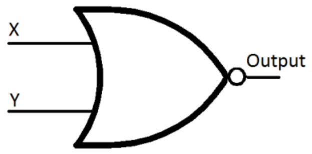
X	Y	Output
0	0	0
0	1	1
1	0	1
1	1	1

Figure 13: Symbol and truth table for an OR gate

This logic gate can be used to check if one or both Boolean values are 1.

NOR gate

A NOR gate takes two binary input and gives a 1 if none of the inputs are 1, and 0 otherwise. This is illustrated in the truth table below to the right, on the left is the symbol used in schematic to indicate a NOR gate. But in all figures in this thesis use the symbol $\overline{\bigcirc}$.



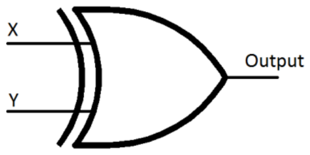
X	Y	Output
0	0	1
0	1	0
1	0	0
1	1	0

Figure 14: Symbol and truth table for a NOR gate

This logic gate can be used to check if both Boolean values are 0.

XOR gate

A XOR gate takes two binary input and gives a 1 if they are different, and 0 otherwise. This is illustrated in the truth table below to the right, on the left is the symbol used in schematic to indicate a XOR gate. But in all figures in this thesis use the symbol ⊕.



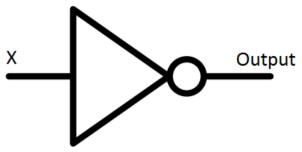
X	Y	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 15: The symbol and truth table for a XOR gate

This logic gate can be used to check if two Boolean values are different.

NOT gate

The NOT gate takes a binary input an output the complement. This is illustrated in the truth table below to the right, on the left is the symbol used in schematic to indicate a NOT gate.



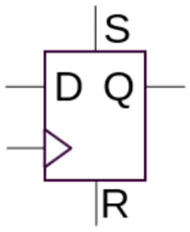
X	Output
0	1
1	0

Figure 16: The symbol and truth table for a NOT gate

This logic gate can be used to compliment a signal.

D flip-flop

A D flip-flop is a component that holds a single binary value Q, that also is its output. For any new clock cycle the value of Q is updated to that of the input D. There are also two other input to the flip-flop S and R, that both are set to 0 for normal use. Below in figure 17, the truth table for a D flip-flop in normal use ($S = R = 0$) is shown on the right. On the left is the symbol used in block schematics. But in so figure in this thesis also uses a simple square to represent a stored it, with inbound arrow for the D signal and an outbound arrow for the Q signal.



Clock	**D**	**Q**
Rising	0	0
Rising	1	1
Not rising	x	Q

Figure 17: Truth table for a D flip-flop in normal use ($S = R = 0$)

By changing the values of S and R it is possible to set the value of Q to 1 or 0, without need to wait for a new clock cycle. In table 2 below is the truth table for the D flip-flop when it is being set to a specific value.

S	**R**	**D**	**Clock**	**Q**
0	1	x	x	0
1	0	x	x	1
1	1	x	x	1

Table 2: Truth table for setting and resetting the output of a D flip-flop

Propagation delay of components

While all of these components can have different propagation delay ranging from 350 ps (picosecond) to 600 ns (nanosecond). This analysis we will use a 5 ns propagation delay for the components AND, OR, NOR, XOR and D flip-flop.

5.4 General concept of the hardware implementation

The main concept is to separate the process of generating a de Bruijn sequence of order n from a de Bruijn sequence of order m into $(n-m)$ independent sub processes called incrementers. Such that each takes a de Bruijn sequence bit stream of order k and calculate the next bit of the $k+1$ order de Bruijn sequence bit stream, and store it and send the previously stored bit to the next incrementer.

Each of the incrementers uses the inverse of the D-homomorphism to find the next bit of the bit stream and complement it if the last $(k-1)$ bit correspond to the selected alternation sequence for the sub process, and shift the input stream according to theorem 10.

Storing one bit and thereby adding a one bit delay for each incrementer, makes it possible to have all the incrementer be independent of each other. So the $(n-m)$ incrementers can be run in parallel, and this allows for a much higher clock frequency than if they where to be computed in a series. Which allow it to produce a higher output of bits per second. This can be further optimize, by minimize the time it take to complete the slowest of the incrementers.

To detect that the last $(k-1)$ output bits from a incrementer correspond to the selected alternating state, the naive approach would be to save the $(k-1)$ last bit and check if they correspond to the selected state. But this take up $(k-1)$ bit of memory, and this will take approximately $\frac{n\cdot(n+1)}{2}$ bit of memory for a $(n-2)$-incrementer.

A more memory efficient way would be to count the number of alternating bit of the bit stream that corresponding to the selected alternating state. This uses $\lceil \log_2 k - 1 \rceil$ bit, and will sum up to approximately $n \cdot \lceil \log_2 n - 1 \rceil$ bit of memory for a $(n-2)$-incrementer. This can also be implemented by any counter module, in this implementation a basic counter is implemented by a primitive *LFSR*. But a counter implemented by a *LFSR* while must have a period greater than $(k-1)$, as the starting state must be different from the state after $(k-1)$ clock of the *LFSR*.

5.5 The hardware implementation

List of signals in an incrementer

- **Reset 1:** Set k-2 until 0 memory cells of the *LFSR* counter.
- **Reset 2:** Set the memory cell $(k-1)$ of the *LFSR* counter.
- **Shift:** Indicate that the counter have reached $(k-1)$, and that the input should be shifted.
- $\mathbf{X_i}$**:** The input stream to the incrementer that is a k-order de Bruijn sequence.
- $\mathbf{Y_i}$**:** The output signal from the incrementer that is a S$(k+1)$-order de Bruijn sequence.
- **Mode:** A binary value that determines the selected alternating sequence. If it is 0 it is $x01$.., and if it is 1 it is $x10$....
- **First state** bit**:** The first bit of the current possible alternating state.
- **P:** If 1, indicates that the current phase is 0.
- $\mathbf{P_-}$**:** If 1, indicate that the current phase is +1.
- $\mathbf{P_+}$**:** If 1, indicate that the current phase is -1.

List of signals that are common for all incrementers

- **Set state:** If 1, then the system will output its current state and set the state of the system to be the input. If 0, then it will have no effect on the system.

The Main module

The binary inverse of the D-homomorphism is:

$$D_k^{-1}(C) = (k, c_0 \oplus k, c_1 \oplus (c_0 \oplus k), ..., c_{2^m-1} \oplus (...),\ k \in \{0,1\}$$

As can seen by this, the sequence generated by the D-homomorphism starts with either a zero or a one. The next bit is the first bit of the input stream *exclusive or* with the previous output bit, and so on. So it is obvious that the previous output bit have to be stored for every module. To implement the joining of the two cycles generated, one of the two alternating sequences have to be chosen. As this is a binary decision, it needs only one bit to be represent in an encoding. The problem is then to detect that the last $(n-1)$ bits of the output stream correspond to the chosen alternating sequences. This can be implemented by a counter that count up to $(n-1)$ if the sequences matches, and reset the counter if it does not.

For the following input bits have to be shifted, and this can be implemented by storing two previous input bit. Details of the module that handle phase shown in the next sub section. The counter and the control module that run the circuit and manages the setting of the register, is presented in the sub sections after that. Below is a block schematic of the main module that performs the D-homomorphism implementation on the input stream X_{i+1} and the shifted input bits X_i, X_{i+2}.

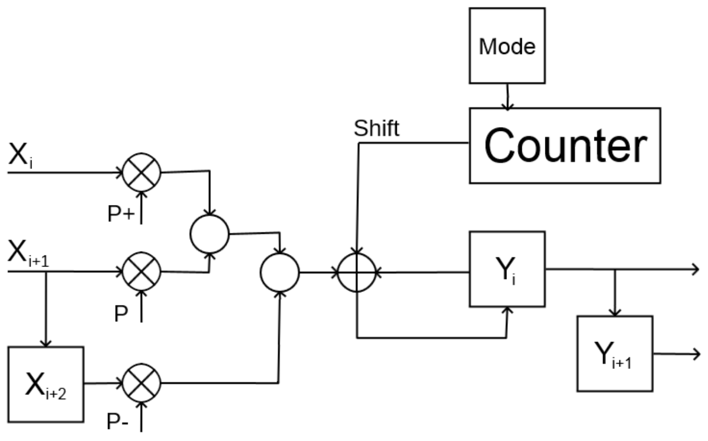


Figure 18: The block schematic of the Main module

The cost of implementing this logic block is:

Number of OR: 2

Number of XOR: 2

Number of AND: 3

Number of Memory cells: 4

Longest path: 2 OR, 1 AND, 2 XOR, 1 D flip-flop. Total of 6 components. So the total cost of this is 11 components for each incrementer, for a total of $11 \cdot (n - m)$ components for a $(n - m)$-incrementer and component complexity is $O(n)$.

The Phase module

P_+, P_- and P are the current values of memory cells, and P'_+,P'_- and P are the next value of the memory cells. In the truth table below it is shown how phase is changes depending on Shift, Mode, First state bit and the current value of the memory cells. Below that in figure 19 is the block schematic of the phase module.

Shift	**Mode⊕First state *bit***	$\mathbf{P_+}$	**P**	$\mathbf{P_-}$	$\mathbf{P'_+}$	$\mathbf{P'}$	$\mathbf{P'_+}$
0	x	x	x	x	$\mathbf{P_+}$	**P**	$\mathbf{P_-}$
1	0	0	1	0	1	0	0
1	0	0	0	1	0	1	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0

Table 3: Truth table for the phase module

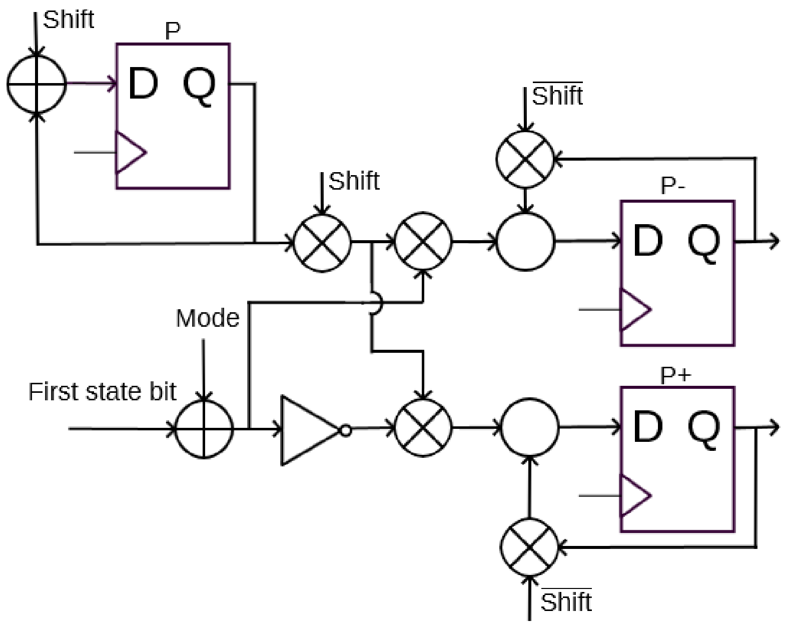


Figure 19: The block schematic of the Phase module

The cost of implementing this logic block is:

Number of NOT: 1
Number of OR: 2
Number of XOR: 2
Number of AND: 5
Memory cells: 3
Longest path: 1 NOT, 1 AND, 1 XOR, 1 D flip-flop. Total of 5 components.

So the total cost of this is 13 components for each incrementer, for a total of $13 \cdot (n - m)$ components for a $(n - m)$-incrementer and component complexity is $O(n)$.

The Set First State bit module

This module is used to store the first bit of any possible alternating state $x01...$ or $x10..$ depending on which is selected. Below in the truth table it is shown how the value of the memory cell changes depending on the values of Reset 1 and Reset 2. Under that in figure 20 is the block schematic for the module.

Reset 1	**Reset 2**	**First state** bit
0	x	**First state** bit
1	0	y_{i-1}
1	1	y_i

Table 4: Truth table for the Set First State bit module

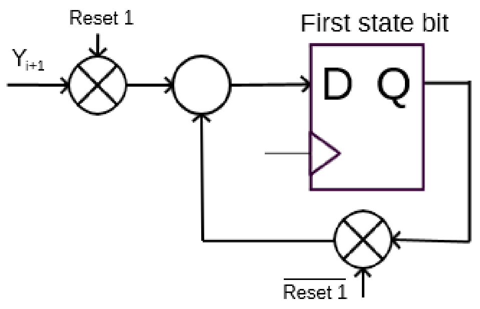


Figure 20: The block schematic of the Set First State bit module

The cost of implementing this logic block is:

Number of OR: 1
Number of AND: 2
Number of Memory cells: 1
Longest path: 1 AND, 1 OR, 1 D flip-flop. Total of 3 components.

So the total cost of this is 4 components for each incrementer, for a total of $4 \cdot (n - m)$ components for a $(n - m)$-incrementer and component complexity is $O(n)$.

The State counter module implemented by a *LFSR*

If the previous output bit and the current output bit are different the counter will increment itself, as this correspond to a part of an alternating sequence. The counter itself will need to count up to $(k-1)$ for an k-incrementer, so the *LFSR* must have to have a period of at least k. Which means that the order of the *LFSR* have to be at least $\lceil \log_2 k \rceil$, and use $\lceil \log_2 k \rceil$ bits of memory.

Next the feedback function should have be as simple as possible, preferably only depend on two state variables (which is minimum for a maximum-length *LFSR*). To detect that the counter has reach $(k-1)$ it will have to look at some of the state variables, and the number of these should be as low as possible.

E.g. for $(k-1) = 5$, the order of the *LFSR* have to be at least 3. The feedback function $f(x_0, x_1, x_2) = x_2 \oplus x_1$ and the initial state $(0, 0, 1)$ will generate the cycle $(0, 0, 1, 1, 1, 0, 1)$ and the state after five clock is $(0, 1, 0)$. To detect this the detecting function $D(x_0, x_1, x_2) = \overline{x}_0 \otimes \overline{x}_2$, which is be true for only this state. As the all zero state does not occur in sequence of length greater than 1, if it is generated by a *LFSR*.

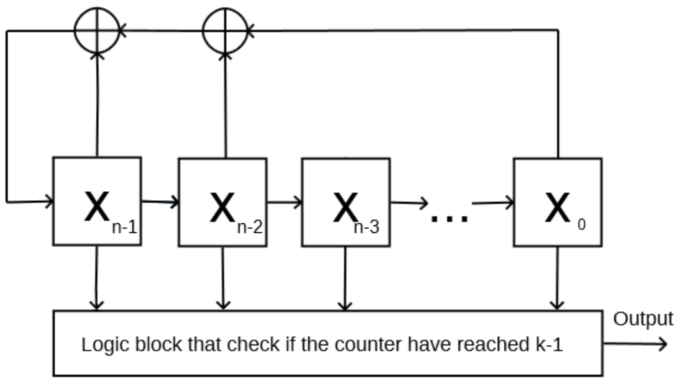


Figure 21: The block schematic of the *LFSR* counter module

To calculate the cost of such a *LFSR* counter, assume that the feedback function depend on two state variables. Then this can be implemented by a single XOR gate. Assume that $\lceil \log_2 k \rceil$ state variables have to be check to detect at the *LFSR* a have reached the $(k-1)$ state. This can be implemented by $\lceil \log_2 k \rceil - 1$ AND gates, and the register will need $\lceil log_2 k \rceil$ memory cells.

The cost of implementing this logic block is:

Number of XOR: 1

Number of AND: $\lceil log_2 k \rceil - 1$

Number of Memory cells: $\lceil log_2 k \rceil$

Longest path: $log_2(\lceil log_2 k \rceil) - 1$ AND, 1 D flip-flop or 1 XOR, 1 D flip-flop. Total of $log_2(\lceil log_2 k \rceil) - 1$ or 2 components.

This gives a component complexity of $O(log_2 k) = O(log_2^2 k)$ and the propagation delay is $log_2(\lceil log_2 k \rceil)$ AND or 2 components as this is the longest part any signal has to travel for every clock. So the time complexity is $O(log_2 k) = O(log_2^2 k)$.

The module to reset the *LFSR* counter

When the two consecutive output bits of the main module are identical (either 00 or 11) the counter will have to be reset. If the two bit are the same as the first bit of the chosen alternating sequence, the counter will have to be reset to 1 as the last bit can then be the first bit of a sequence that is equal to the chosen alternating sequence. But if it is not, then the counter will have to be reset to zero. So the difference between those two type of rest are only the value of memory cell $(n-1)$. This implement by the two reset signals $Reset\,1$, and $Reset\,2$.

When the logic block that check if the counter has reached step $(k-1)$, the cells $(n-2)$ until 0 is set to 1. Cell $(n-1)$ is set to 1 if the current bit can be the first bit of the selected alternating sequence, and set to 0 otherwise. This can be implemented by a small logic block that checks if the two subsequent bits are the same, and if they are it checks if the last bit can be the first bit of the selected alternating sequence. Below the logic block for this is shown.

Note that the signal $Reset\,1$ is stored in a memory cell and is used to reset the memory cells of the counter in the same clock cycle that it is stored. If the $Reset\,1$ memory cell that stores $Reset\,1$ dont́t have a strictly faster propagation delay than the memory cells of the counter, it can also reset the counter at the start of the following clock cycle. This problem may also be fixed by changing the implementation, either by using other types of components or redesigning the way the resetting of the counter works.

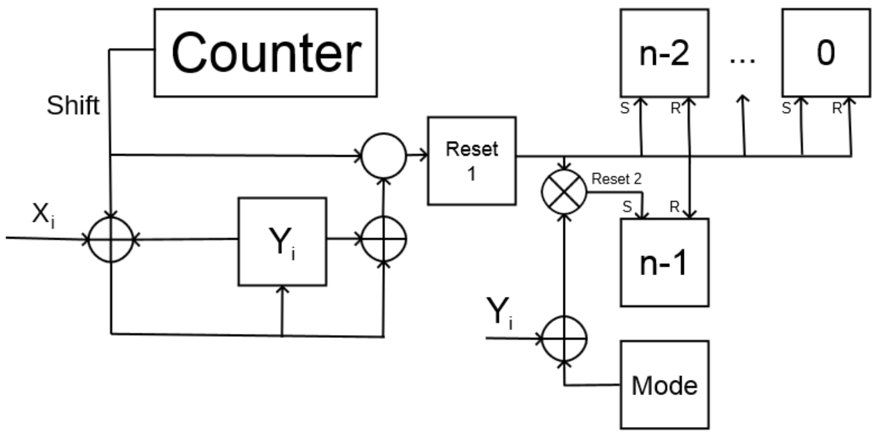


Figure 22: The block schematic for the module to reset the counter

The cost of implementing this logic block is:

Number of OR: 1
Number of XOR: 2
Number of AND: 1
Memory cells: 1
Longest path: 1 XOR, 1 OR, 1 D flip-flop or 1 XOR, 1 AND, 1 D flip-flop. Total of 3 components.

So the total cost of this is 5 components for each incrementer, for a total of $5 \cdot (n - m)$ components for a $(n - m)$-incrementer and component complexity is $O(n)$.

The module for setting and retrieving the state of the $(n-m)$-incrementer

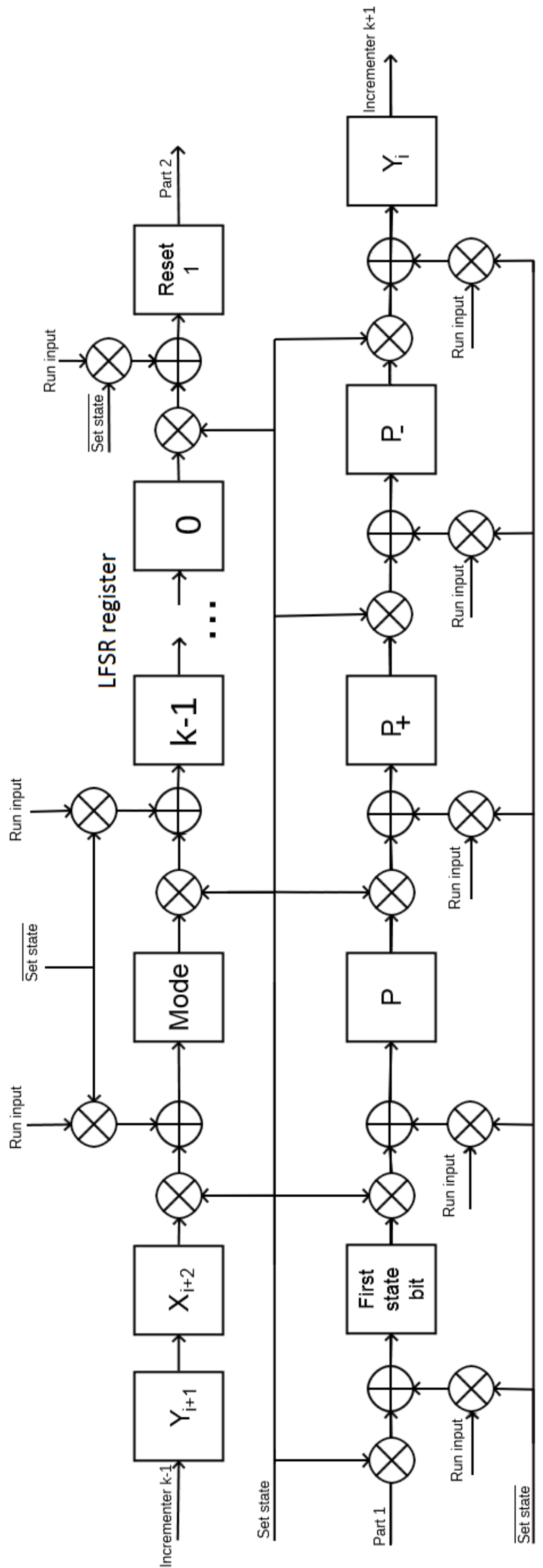
The problem of retrieving the information stored in all the *D flip-flop* can be solved by having them be connected by an AND gate, that has as input the previous memory cell and a control signal *Set state*. So if $Set\,state = 1$ the bits are then shifted out one at a time, and a new state can be simultaneously shifted in. To make sure that the values of the memory cells are only shifted and not changes, the normal processes of the *FSR* have to be stopped. So the normal processes must have the requirement that $Set\,state = 0$, use $\overline{Set\,state}$ to refer to the complement of *Set state*.

The bits Y_i,$Y_{i+2}$and X_{i+2} are already shifted. The same with the memory cells of the LFSR register.This process can be implemented by using 2 AND and 1 Xor gate before any memory cells which input is not simply a shift from another memory cell. The original input goes through an AND gate that is open if $Set\,state = 1$, and the input from the shift goes through an AND gate that is open if $Set\,state = 0$. Both signals are then sent to a XOR gate, and the output of this is then sent to the memory cell. The schematic of this is shown on the next page, the signal "Run input" is the original input signal.

The cost of implementing this logic block is:

Number of XOR: 8
Number of AND: 16

So the total cost of this is 24 components for each incrementer, for a total of $24 \cdot (n - m)$ components for a $(n - m)$-incrementer and component complexity is $O(n)$.


LFSR register
Incrementer k-1
Y_{i+1}
X_{i+2}
Mode
k-1
...
0
Reset 1
Part 2
Run input
Set state
Set state
Part 1
First state bit
P
P+
P-
Y_i
Incrementer k+1

Staring the $(\mathbf{n} - \mathbf{m})$-incrementer for an initial state

Since every incrementer need two bit of the input stream to be able to start running (X_{i-1} and X_i), the clock signal to the different incrementers memory have to be delayed by 2 clock cycle for every incrementers. So an $(n-m)$-incrementers need to clock $2 \cdot (n - m)$ times before it can being to output the final de Bruijn sequence of order n.

This operation can be performed by a software program, to reduce the number of required components, a hardware solution is not implemented in this thesis. As this is only a one time operation, that can be relatively efficiently implemented by a software application.

5.6 Correctness

To prove the correctness of this implementation, that it do produce a unique de Bruijn sequence for all valid configuration of the device. Start by looking at the most simple iteration.

Given an m order de Bruijn sequences incrementer, and let the output be of order $(m + 1)$. By selecting one of the two alternating sequences, one of two possible de Bruijn are produces. Then as long as the incrementer can detect the selected alternating state and complement the next output bit. The output sequence will be a de bruijn sequence of order $(m + 1)$.

By adding another incrementer that takes this $(m + 1)$ order de Bruijn sequence as an input, and output a $(m + 2)$ order de Bruijn sequence. So if the implementation works for a simple incrementor, it can work for an $(n - m)$-incrementor. As it is only serie of simple incrementers. This produces a de Bruijn for any of order, given enough incrementors. This given that all the modules are correctly implemented.

Number of elements

The main modules:

Number of OR: 2

Number of XOR: 2
Number of AND gates: 3
Number of Memory cells: 4

The *LFSR* counters:
Number of XOR: 1
Number of AND gates: $\lceil log_2 k \rceil - 1$
Number of Memory cells: $\lceil log_2 k \rceil$

The Resetting of the *LFSR* modules:
Number of OR: 1
Number of XOR: 2
Number of AND gates: 1
Number of Memory cells: 1

The Phase modules:
Number of OR: 2
Number of XOR: 2
Number of AND gates: 5
Number of NOT: 1
Number of Memory cells: 3

The Set first sate bit module:
Number of OR: 1
Number of AND gates: 2
Number of Memory cells: 1

The setting and retrieving of all the values of the memory cells:
Number of XOR: 8
Number of AND gates: 16

An incrementer:
Total number of OR: 6
Total number of XOR gates: 15
Total number of AND gates: $\lceil log_2 k \rceil + 26$
Total number of NOT gates: 1
Total number of memory cells: $\lceil log_2 k \rceil + 9$
Total number of components: $2 \cdot \lceil \log_2 k \rceil + 57$

Let $Comp_{Total}$ be the number of components for an $(n-m)$-incrementer, then:

$$Comp_{Total} = \sum_{i=m}^{n} (2 \cdot \lceil \log_2(i) \rceil + 57) = 57 \cdot (n-m) + 2 \cdot \sum_{i=m}^{n} \lceil \log_2(i) \rceil$$

It is easy to see that:

$$\sum_{i=m}^{n} (\log_2(i)) \leq \sum_{i=m}^{n} \lceil \log_2(i) \rceil < \sum_{i=m}^{n} (\log_2(i) + 1) \ (1)$$

Since:

$$\sum_{i=m}^{n}(\log_2(i)) = n \cdot log_2(n) - m \cdot log_2(m) - n + m \ (2)$$

$$\sum_{i=m}^{n}(\log_2(i)+1) = n \cdot log_2(n) - m \cdot log_2(m) \ (3)$$

Substituting (2) and (3) into (1) gives:

$$n \cdot log_2(n) - m \cdot log_2(m) + 55 \cdot (n-m) \leq Comp_{Total} < \\ n \cdot log_2(n) - m \cdot log_2(m) + 57 \cdot (n-m)$$

Based on this the component complexity of a $(n-m)$-incrementer is $O(n \cdot log_2 n)$ if m is a constant.

Propagation delay

We add the delay for the D-flip flops at the end. In a circuit, the clock frequency is limited by the longest path any signal have to travel for each clock cycle. For the counter *LFSR* this is time for the feedback function components to stabilize on the next bot of the register. $\lceil log_2 \lceil log_2 n \rceil \rceil$ AND are needed to check if the new state of the counter *LFSR* have reached the $(n-1)$ state.

The signal for this (*Shift*) is used by the main module, the phase module and the module to reset the counter, so all these modules have to wait $\lceil log_2 \lceil log_2 n \rceil \rceil$ AND before it can start to calculate its output values.

The main module have a constant delay of 2 OR, 1 AND then this signal is XORed with *Shift* and another signal. For the phase module the maximum propagation delay is the maximum is *Shift*, 2 AND, 1 OR. For the resetting module the maximum propagation delay is the maximum is Shift, 1 OR, so it is strictly lower than the phase module and can therefore be ignored.

The set first sate bit module only have a constant delay that is lower than the constant delay in the main module, so it can also be ignored. So the maximum propagation delay is the maximum of 2 OR, 1 AND, 2 XOR or $\lceil log_2 \lceil log_2 n \rceil \rceil$ AND, 1 XOR or $\lceil log_2 \lceil log_2 n \rceil \rceil + 2$ AND, 1 OR.

For n $<$ 5 the total propagation delay is:

2 OR, 1 AND, 2 XOR, 1 D flip-flop.
Which is a total of 6 components.

For n $\geq$ 5 the total propagation delay is:

$\lceil log_2 \lceil log_2 n \rceil \rceil + 2$ AND, 1 OR, 1 D flip-flop.
Which is a total of $\lceil log_2 \lceil log_2 n \rceil \rceil + 4$ components.

So the time complexity to generate a bit is $O(log_2^2 n)$, and if we assume a propagation delay of 5 ns for each components. The total propagation delay is $5 \cdot (\lceil log_2 \lceil log_2 n \rceil \rceil + 4)$ ns for $n \geq 5$.

Table for cost and efficiency

Below is a table that illustrate how the cost of component and memory grows depending on the order of the de Bruijn sequences generated and the number of mega Byte per second (MB/s) generated. These numbers are for an $(n-2)$-incrementer, with 5 ns propagation delay for each components.

Order	**Comp.**	**Comp./Ord.**	**Mem.**	**MB/s**
75	4989	66.5	1059	3.41
150	10395	69.3	2282	3.41
300	21507	71.7	4877	2.98
600	44332	73.9	10366	2.98
1200	91183	76.0	21934	2.98
2400	187286	78.0	46296	2.98
4800	384293	80.1	97401	2.98
9600	787908	82.1	204410	2.98
100000	8883441	88.8	2468925	2.65

Table 5: Table that illustrate the cost and efficiency to generate de Bruijn sequences

As can be seen form the table, the efficiency of the implementation is decreasing, as the cost of memory cells/components increases by more that double when the order doubles.

5.7 Analyze of the final product

Pros

Predictable number of components, and throughput of bits per second. The time complexity of generating a bit is $O(log_2^2 n)$. So there is measurable and predictable performance level that can be used to generate 2^{n-m} unique de Bruijn sequences for every de Bruijn sequence of order m. A reduction in propagation delay of the AND, XOR or D flip-flop leads to an increase in bit throughput. Can be used in addition to other methods, as the principle allows the generation to be used from any order of de Bruijn sequences. This makes it possible to extend de Bruijn sequences of any order.

Cons

Need to have a hardware implementation to be efficient at all. The numbers of memory cells and components grows $O(n \cdot \log_2 n)$. The initial state have a complexity of $O(n \cdot \log_2 n)$ bits. To get current state of the $(n-m)$-incrementer, $O(n \cdot \log_2 n)$ clock cycle are needed to retrieve it. $2 \cdot n$ clock cycles have to be used to initialize a starting configuration. This can be done on different device beforehand, or in a software application. The module to reset the counter can be improved.

Hardware vs software

The difference in time and space/component complexity for the hardware and software is shown below in table 6.

	Software	**Hardware**
Time complexity to generate a bit	$O(n)$	$O(log_2^2 n)$
Space/component complexity	$O(n \cdot log_2 n)$	$O(n \cdot log_2 n)$

Table 6: Table show the difference between this hardware implementation and a general software implementation

Testing

For every start configuration at least $k \cdot n$ should be generated, for some constant k. The resulting output sequence should be tested try to find the from this sub sequence. The worst case is that the sequence generated is de Bruijn sequence that can be found by joining the two cycles of a *LFSR* with a maximum period. Which can be found by using the Berlekamp-Massey algorithm on at most $(2 \cdot n)$ bits. Of course this may fail if the 0 bit from the all zero cycle of the *LFSR*, is a part of the $(2 \cdot n)$ bit the algorithm is run on. So this should be taken into consideration when testing the sequence.

The same is true if the de Bruijn sequence is complemented sequence of a de Bruijn sequence that can be found by joining the two cycles of a *LFSR* with maximum period. Which can be found the same way by running the complemented bit. The reversed complement is also easy to find, so it should be checked to make sure that the sequence at least cannot be found by simply using the Berlekamp-Massey algorithm.

Usability

As the devices simply takes a de Bruijn sequence stream as an input, it can be combined with different technique that are able to efficient generate de Bruijn sequences as long as there is enough memory, e.g. Annexstein [1]. This sequence can then the input, this reduces the total number of components. The device can also be more generalized, by making it possible to have multiple input and output points. So that the order on the input sequence and output sequence can be changed. Also possible to add complementing of any of the orders.

Conclusion

The main thing I have been working on

I have created a method to apply Lempels inverse D-homomorphism on a bit stream, without having to store the current state. That takes any de Bruijn sequence S of order m and binary sequence s of length $(n - m)$ and generates unique a de Bruijn order n for each unique pair of S and s. For a total of $2^{2^{m-1}+n-2\cdot m}$ unique sequences.

This was implemented in a simple hardware circuit, that has a time complexity to generate a bit of $O(log_2^2 n)$ a component complexity of $O(n \cdot log_2 n)$. This is relatively good scales time and space/component complexity, and the cost in time and components are predictable.

Other things I have been looking at

Method to find de Bruijn by looking for cross-join pairs

Thought of a method to find new de Bruijn functions of order n from a de Bruijn functions of order n. The first step is to search for i subsequent bits that differ by only one bit t from a possible cycle in B_n. So if the entry in g that determines this bit t is swapped, and this does not change any of the other bit. Then de Bruijn cycle is split into two cycles of length $2^n - i$ and i. Since we now all the states of the cycle with i bits, it is easy to find out if one or more of these states s have a possible successor in the cycle of length $2^n - i$.

If there is such a state, then by changing the value of truth table for g for the entry that changes the successor of s. This will join the two cycles in a new de Bruijn sequence, if $t \neq s$.

Note that in a cycle each state have only one successor and one predecessor, so for a cycle of length i. Start with a state s, if both possible successor are in the cycle, then the number of possible successors are $i - 2$. Then take the next state that has not have its possible successor looked at and continue. Since the number of possible successor is lowered by 2 for each check that does not find a join-state is not found. If i is an odd number, the cycle is guarantied to contain at least one state that can be used to join the two cycles into a de Bruijn sequence, such that $t \neq s$.

Did not find a way to construct a reliable algorithm, such that this efficiently implemented.

de Bruijn sequences over non-binary alphabets

Did some basic works on de Bruijn sequences over alphabet larger than 2 and the relation between them and binary ones. No real progress was made.

New methods to generate new de Bruijn sequences

Did work on a new method to find de Bruijn sequences from de Bruijn of lower orders. But did not find a new method.

New methods to generate new de Bruijn sequences from others of the same order

Did work on a new method to find de Bruijn sequences from de Bruijn of the same orders. Did find some basic results, but nothing new.

Thing I would have like to do if I had more time

- Would like to improve or redesign parts of the hardware implementation that I was not satisfied with. The way the counter module is being rest, or the module to change or retrieve the current state of the $(m - n)$-incrementer could have been improved.
- Physically make a prototype of a $(n - m)$-incrementer, for a relatively small n to get more practical data on how to optimize the process.
- Try to find ways to merge multiple subsequent incrementers to make it possible to optimize the implementation.
- Try to deduce bounds on linear complexity, distribution and correlation of the sequences that can be generated by $(n - m)$-incrementer.

Subjects that I would like to look into

- The properties of modified de Bruijn sequences especially type 1. And ways to use this with $(n - m)$-incrementers.
- Looking for more and efficient ways to generate binary de Bruijn sequences that scales well for very large orders.
- Try to find ways to efficiently generate de Bruijn sequences of high orders over non-binary alphabets.

References

[1] F. S. Annexstein, “Generating de Bruijn sequences: AN efficient implementation,” *IEEE Trans. Comput.*, vol 46, no. 2, pp. 198–200, Feb. 1997.

[2] N. G. de Bruijn, “A combinatorial problem”, *Koninklijke Nederlandse Akademie van Wetenschappen*, vol 49, no. 1, pp. 758–764, 1946.

[3] A. H. Chan, R. A. Games, E. L. Key, “On the complexities of de Bruijn sequences,” *J. Combin. Theory*, Ser A, vol. 33, pp. 233–246, Nov. 1982.

[4] T. Chang, B. Park, Y.H. Kim, I. Song, “An efficient implementation of the D-homomorphism for generation of de Bruijn sequences,” *IEEE Transactions on Information Theory* 45, pp. 1280–1283, May 1999.

[5] E. Dubrova, “A Transformation From the Fibonacci to the Galois *NLFSR*s”. *IEEE Transactions on Information Theory*, vol. 55, no. 11, Nov. 2009.

[6] E. Dubrova, *Sequences and Their Applications* – SETA 2010.

[7] E. Dubrova, *Des. Codes Cryptogr.* 2014.

[8] C. Flye Sainte-Marie, “Solution to question nr. 48,” *I’lntermediuire des iathematiciens*, pp. 107–110, 1894.

[9] H. Fredricksen, “A Survey of Full Length Nonlinear Shift Register Cycle Algorithms,” *SIAM Review 24*, no. 2 , pp. 195–221, Apr. 1982.

[10] R. Games, “A generalized recursive construction for de Bruijn sequences,” *IEEE Transactions on Information Theory*, vol. 29, no. 6, pp. 843–850, Nov. 1983.

[11] S. W. Golomb, *Shift Register Sequences*, Holden-Day, San Francisco, CA, 1967.

[12] T. Helleseth, T. Klove, “The number of cross-join pairs in maximum length linear sequences,” *IEEE Transactions on Information Theory*, vol. 37, no. 6, pp. 1731–1733, Nov. 1991.

[13] K. Kjeldsen, “On the cycle structure of a set of nonlinear shift registers with symmetric feedback functions,” *J. Combinatorial Theory*, Ser. A., 20 , pp. 154–169, Mar. 1976.

[14] A. Lempel, “On a homomorphism of the de Bruijn graph and its applications to the design of feedback shift registers”, *IEEE Trans. Comput.*, vol 19, no. 12, pp. 1204–1209, Dec. 1970.

[15] C. Li, X. Zeng, C. Li, T. Helleseth, “A Class of de Bruijn Sequences,” *IEEE Transactions on Information Theory*, vol. 60, no. 12, pp. 7955–7969, Dec. 2014.

[16] C. Li, X. Zeng, C. Li, T. Helleseth and M. Li, “Construction of de Bruijn Sequences From *LFSR*s With Reducible Characteristic Polynomials,” *IEEE Transactions on Information Theory*, vol. 62, no. 1, pp. 610–624, Jan. 2016.

[17] J. Massey, "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, Jan. 1969.

[18] G. L. Mayhew, S. W. Golomb, "Linear spans of modified de Bruijn sequences," *IEEE Transactions on Information Theory*, vol. 36, no. 5, pp. 1166–1167, Sep. 1990.

[19] J. Mykkeltveit, "A proof of Golomb's conjecture for the de Bruijn graph," *J. Comb. Theory*, Ser. A, 13, pp. 40–45, Aug 1972.

[20] E. Selmer, *Linear Recurrence Relations over Finite Fields*, Dept. of Math., University of Bergen, 1966.

[21] J. Søreng, "Symmetric shift registers," *Pacific J. Math.*, 85 , pp. 201–229, Nov. 1979.

[22] J. Søreng, "Symmetric shift registers. II," *Pacific J. Math.*, 98, pp. 203–234, Jan. 1982.

[23] W. Zheng, Y. L. Cao, Y. C. Zhou and T. Y. Xu, "A generalization of modified de Bruijn sequences," The 2nd International Conference on Information Science and Engineering, Hangzhou, China, pp. 1705–1708, 2010.

[24] N. Zierler, "Linear recurring sequences," *J. Soc. Indust. Appl. Math.* 7, pp. 31–48, 1959.