# Shift Register Based Applications with MVL Switching Functions

A Tutorial of an IP Portfolio

Peter Lablans
Ternarylogic LLC
Morristown, NJ, USA
www.ternarylogic.com

**Abstract— MVL shift register based applications include scramblers/descramblers, error correcting coders and symbol sequence generators. Examples are provided how to create implementations of specific shift register based applications, by focusing on a desired outcome and by applying specific n-state switching table properties.**

***Keywords—nonbinary feedback shift registers, n-state switching functions, n-state inverters, scrambling, sequence generation, error correcting coding***

## I. Introduction

Dr. G. Blaauw [1, 2] distinguishes 3 hierarchical levels in computer design: (1) the architecture (what the user sees); (2) the implementation (the logic design); and (3) the realization (physical components). The implementation level is somewhat of a given nowadays in binary technology. Most functions are available in IC libraries. Furthermore, most binary logic designs use a limited set of components, such as AND, NAND and XOR gates. Novel switching technology (the realization level) may have nonbinary switching capabilities. Converting binary logic designs to nonbinary logic designs (the implementation) is generally a non-trivial matter. In most cases a completely new design is required. An example of first creating a general switching model and then applying that model to find the implementation for any state memory latches is provided in [3]. Shift register based circuits and applications are among the most widely applied binary implementations and include: sequence generators, scramblers and descramblers, and error correcting code generators (including Reed Solomon Codes and convolutional codes). One critical element of design of nonbinary shift register applications is the use of reversible (invertible) nonbinary switching functions.

## II. Reversible Nonbinary Switching Functions

### A. 4-state switching functions

Nonbinary switching functions can be expressed as (1) a single variable operation f(x) with x able to assume one of n states with n>2. Such an implementation function will be called an inverter or n-state inverter. An n-state inverter *inv*(a) transforms a single n-state input variable *a* into a single n-state output *c*. A multi-input function *sc*(*a*,*b*) with *a* and *b* each *n*-state variables with *n*>2, generates an *n*-state variable *c* as illustrated in FIG.1. The inverter is determined by a 1-dimensional array. The 2-input device is determined by a 2-dimensional array or switching table.

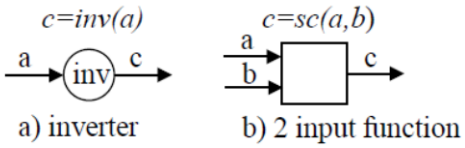


FIG. 1 N-state switching functions

### B. 4-state reversible switching tables

For illustrative purposes most of the switching tables provided herein will be 4-state (*n*=4). The reason for this is that it allows the use of at least somewhat known switching tables, in particular an addition and multiplication over finite field *GF*(4), which is a binary extension field. One example of a 4-state reversible inverter is [0 1 2 3]→[0 2 3 1], which is also a multiplier 2 over *GF*(4).

The 4-state tables as provided in FIG. 2, represent 4-state 2 input/single output functions. The switching tables of $GF_{4+}$ and $FR_4$ are reversible and commutative. $GF_{4+}$ and $GF_{4*}$ define *GF*(4) and are associative and distributive, which makes symbolic manipulation easier. $GF_{4+}$ and $FR_4$ are both self-reversing in the sense that a first input state can be recovered commutatively from the generated output state and a second input state. These functions are thus comparable, in a way, to the binary "≠" (XOR) and the binary "=" (EQUIVALENT) function. However, $FR_4$ is not associative.

| $GF_{4+}$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 0 | 3 | 2 |
| 2 | 2 | 3 | 0 | 1 |
| 3 | 3 | 2 | 1 | 0 |

| $GF_{4*}$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 |
| 2 | 0 | 2 | 3 | 1 |
| 3 | 0 | 3 | 1 | 2 |

| $FR_4$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 3 | 2 | 1 | 0 |
| 1 | 2 | 1 | 0 | 3 |
| 2 | 1 | 0 | 3 | 2 |
| 3 | 0 | 3 | 2 | 1 |

FIG. 2 4-state switching tables

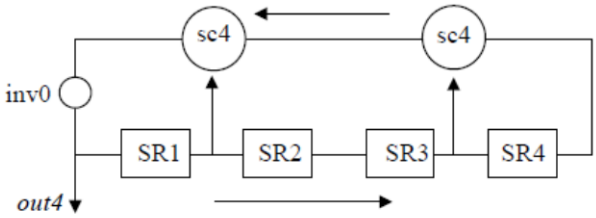


FIG. 3 A 4-state PN sequence generator

## III. Shift Register Based Circuits

In the following sections, implementations of 4-state shift register based circuits/applications are provided, which include sequence generators, sequence scramblers and descramblers, sequence detectors and generator synchronization.

### *A. 4-state pseudo-noise (PN) sequence generators*

FIG. 3 illustrates a 4-state shift-register with feedback based sequence generator of a pseudo-noise (PN) sequence *out4* of 4-state symbols. The shift register has register elements [SR1 SR2 SR3 SR4] and is in a Fibonacci configuration (as opposed to a Galois configuration). The content of the register elements is shifted upon a clock signal, which is not shown but is assumed. All functions are evaluated before the register content is shifted. The feedback function '*sc*4' is the function $GF_{4+}$ as illustrated in FIG. 2. The inverter *inv*0 is the inverter [0 1 2 3]→[0 2 3 1] and is a multiplication 2 over *GF*(4). The length of the PN sequence is $(n^k\text{-}1) = (4^4\text{-}1)=255$ 4-state symbols (with *n* the number of states and *k* the number of shift register elements.)

Another 4-state PN generator of 255 4-state symbols is shown in FIG. 4. The function *rc4* is the function $FR_4$ shown in FIG.2. The inverter *inv2* is [0 1 2 3]→[0 1 3 2] and *inv3* is [0 1 2 3]→[ 3 0 1 2].

Both generators generate a PN sequence (*out4* and *out42*) of 255 4-state symbols. Pseudo-randomness can be determined from a modified auto-correlation of the sequence of 255 symbols. The (auto)-correlation is determined by calculating a sum of the sequence against a shifted version wherein a fixed value is added to the sum when a symbol in the sequence and a symbol in the shifted sequence are identical and nothing is added or a fixed value is subtracted when symbols are not identical. The correlation graphs are shown in FIG. 5.

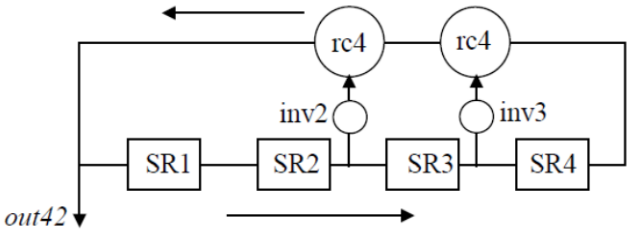


FIG. 4 Another 4-state PN sequence generator

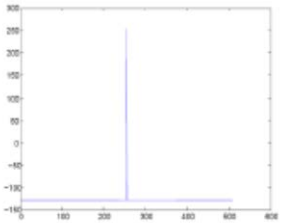

FIG. 5a Auto-correlation of *out4* and *out42*

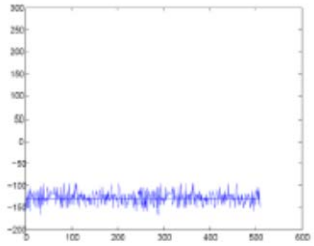

FIG. 5b Cross-correlation between *out4* and *out42*

The modified auto-correlation graphs in FIG. 5a of *out4* and *out42* are identical and resemble the graphs of binary PN sequences. The cross-correlation graph of the two sequences in FIG. 5b shows that the sequences *out4* and *out42* are different and are not shifted versions. [4].

A true PN n-state sequence of $(n^k\text{-}1)$ n-state symbols has an auto-correlation graph as in FIG. 5a. The known, unmodified, auto-correlation will create side-lobes in the graph, making it less directly visible if a sequence is PN. Another way to determine if a sequence generator is a PN generator is to analyze the content of the shift register [SR1 SR2 SR3 SR4] during sequence generation. If [SR1 SR2 SR3 SR4] is unique for each generated symbol then the generator is PN. [4].

### *B. Scramblers and Descramblers with Sequence Generators*

The simplest scrambler of a source of *n*-state symbols is a combination of the source with a sequence generator through a reversible *n*-state switching function. In one example the switching function that combines the two streams is a self-reversing function. Such a scrambler is illustrated in FIG. 6. The result *out* can be described by *out*=*rc*4(*in*,*seq*).

The descrambler corresponding is the same set-up of FIG. 6 with *in* now being the generated scrambled sequence *out* which has to be descrambled against sequence *seq*. The result is *res*=*rc*4(*out*,*seq*) or *res*=*rc*4((*rc*4(*in*,*seq*)),*seq*)=*in*. The sequence generator of the descrambler must be synchronized with the one at the scrambler. It will be explained later how they can be easily synchronized. Any reversing combination of a reversible scrambling function and a corresponding descrambling function can be used.

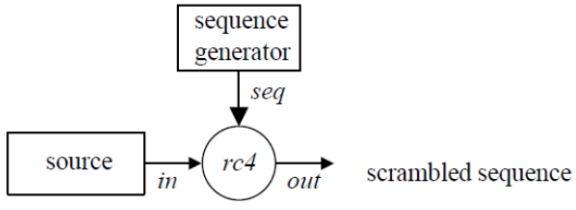


FIG. 6 Scrambler based on sequence generator

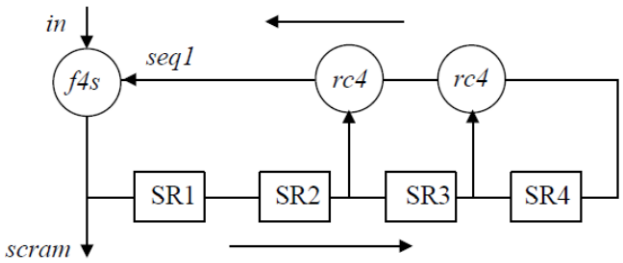


FIG. 7a A 4-state scrambler

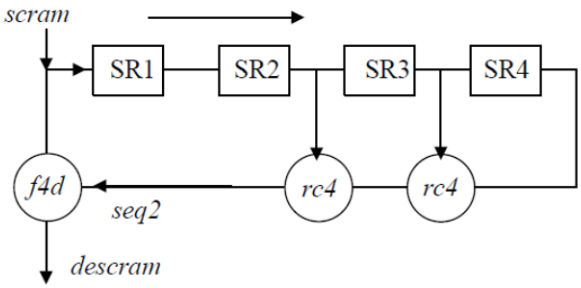


FIG. 7b A 4-state descrambler

*C. Scramblers and Self-Synchronizing Descramblers*

The binary self-synchronizing shift register based descrambler in Fibonacci configuration is known and was developed at Bell Labs. [5] The basis for this descrambler is that one or more errors occur on a line signal during transmission. The descrambler may lose its synchronization with the scrambler as a result. The self-synchronizing descrambler flushes the errors through the shift register and will start to descramble correctly after the errors have been flushed.

Feedback shift registers in Galois and Fibonacci configuration differ in the aspect of timing. Because all functions have to be evaluated before the content of the shift registers are shifted, the Fibonacci configuration with multiple switching functions may require a longer time before a shift can take place. Scramblers/descramblers in Galois configuration are described in [6]. FIG. 7 shows a 4-state scrambler and a corresponding shift register based self-synchronizing descrambler in Fibonacci configuration.

The scrambler has a 4-state feedback shift register with 4 shift register elements [SR1 SR2 SR3 SR4] each enabled to hold and shift signals representing a 4-state symbol. Feedback takes place via taps from shift register elements into 4-state functions *rc*4. The 4-state function *rc*4 in this case is reversible. Also irreversible functions may be used. Also different functions may be applied for each of the feedback functions, including non-commutative functions. A to be scrambled sequence *in* of 4-state symbols is entered on an input of a reversible function *f4s* and combined with a sequence of symbols *seq*1 generated by the feed backshift register. A sequence of scrambled 4-state symbols *scram* is outputted on an output and is also fed back into the shift register.

For descrambling the scrambled sequence *scram* is entered into the descrambler to a 4-state function *f4d* which reverses *f4s*. The function *f4s* may be a self-reversing function, in which case *f4s* and *f4d* are identical. The function *f4s* may be a mod-4 subtraction, thus determining that *f4d* is a mod-4 addition. The sequence *scram* is also entered into the shift register of the descrambler, which is in a forward configuration and not in feedback and which generates a sequence *seq2* which is also provided on an input to *f4d*.

One can easily determine that *seq1* and *seq2* are identical if the scrambler and descrambler both start operating with the identical content of [SR1 SR2 SR3 SR4]. At the first clock pulse a 4-state symbol *in*1 is provided at the input of the scrambler, and a first scrambled symbol *scram*1 is provided in accordance with:

*seq*1= *rc4*{SR2,*rc4*(SR3,SR4)}; and

*scram1* = *fs4*(*in*1, *seq1*).

The descrambler receives the symbol *scram1* and provides:

*seq2* = *rc4*{SR2,*rc4*(SR3,SR4)}; and

*descram1* = *fd4*(*scram1*, *seq2*).

Because *seq1* = *seq2*, the outcome will be *descram1* = *in1*, which is the original input symbol. It is initially simpler to use for *f4s* a self-reversing function that is commutative. In that case *f4d* = *f4s*. In case of a self-reversing function, if *scram1=f4s*(*in1*,*seq1*) then *in1=f4s*(*scram1*,*seq1*) and because *seq2=seq1*, one can see that *descram1=f4s*(*scram1*,*seq2*) and thus *descram1=f4s* (*scram1*,*seq1*)=*in1*.

The descrambler will fail to correctly descramble if the contents of the shift registers of the scrambler and descrambler are not identical, for instance because of a line error in the signal. However, because the shift register of the descrambler is in a forward configuration, no feedback occurs and an error will not propagate in the shift register, but will be flushed and if no further errors occur, the descrambler will automatically re-synchronize with the scrambler and descramble correctly.

The terms scrambler and descrambler are relative terms. The descrambler can be used as a scrambler and the scrambler is then the corresponding descrambler. The descrambler then has a shift register in feedback configuration and an error will be fed back into the shift register and propagate throughout any descrambling result. This is actually a more secure configuration, but requires for the descrambler a facility to determine and re-establish synchronization.

*D. Convolutional Coders/Decoders*

The properties of descramblers and scramblers can be advantageously used in n-state convolutional coders, wherein the coder is for instance formed by two different descramblers

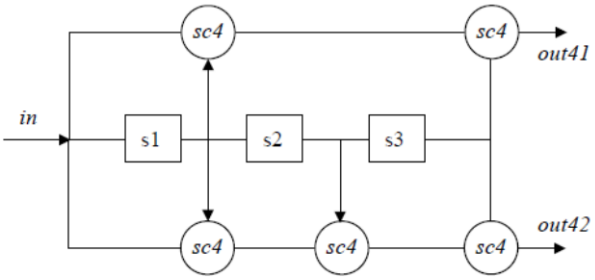


FIG. 8 An n-state convolutional coder

in a non-recursive configuration with *k* shift register elements. This is shown in FIG. 8, wherein the coders are 4-state coders in forward configuration with function *sc4* being an adder over *GF*(4) and each coder using a shift register of 3 elements. As result of an input sequence *in* the two sequences *out41* and *out42* are generated. A decoder in that case can be two corresponding scramblers, which should generate two identical decoded symbol streams. The corresponding decoders are shown in FIG. 9.

Errors in the coded signal will cause two different decoded sequences, indicating that errors have occurred. These errors will propagate with every decoding step. It may be assumed that only a limited number of errors has occurred and that the coded sequences are error free at a certain stage. When one has at least 3 error free consecutive coded symbols in both coded streams, then the correct content of the shift register can be calculated. Furthermore, the content of the shift register reflects previously correctly decoded symbols and thus calculating the shift register content is identical to error correction. Based on an assumption of error-free transmission from a certain point forward, the correct state of the content of the shift registers can be calculated and up to *k* errors can be corrected in the received data stream from the moment of calculating the content. [7]

In this example the function *sc4* of FIGS. 8 and 9 is the addition over *GF*(4) which may be represented by '+.' One should be aware that *a*+*b* over *GF*(4) in this case is the same as *a*-*b*. The content of the shift register [*s*1 *s*2 *s*3] at moment *i* is provided by:

$s1 = out41(i+1) + out42(i+1)$

$s2 = out41(i) + out42(i)$

$s3 = out41(i) + out41(i+1) + out42(i+1) + out41(i+2) + out42(i+2)$

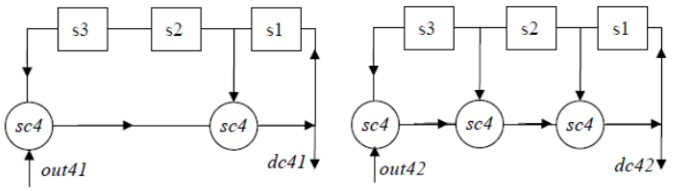


FIG. 9 N-state convolutional decoders

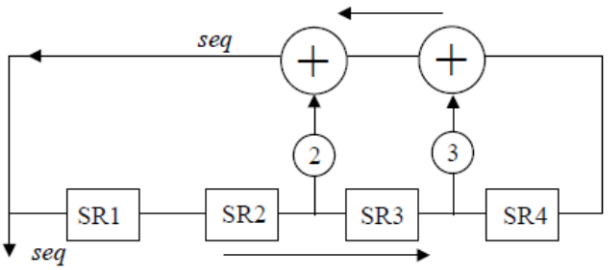


FIG. 10 A 4-state sequence generator

*E. Detectors of Sequences*

There are different circumstances that may require sequence detection. For instance, several sequences may be transmitted at the same time and are superimposed on each other. A presence of a specific sequence has to be detected in the superimposed sequences. In that case a classic correlation approach may be appropriate.

One may also apply a specific sequence such as a PN sequence to mark a place in large amounts or sequence of data. In that case, a continuously running modified self-synchronizing descrambler can easily detect a specific sequence. Assume that a sequence generated by the sequence generator of FIG. 10 is used to mark a location in data that is read from a memory or hard drive. The generated sequence is *seq*. The '+' function is the addition over GF(4) and the inverters '2' and '3' in the feedback taps are the multipliers 2 and 3 over *GF*(4). The first 3 4-state symbols and the following content of the shift register after initial state [R1 R2 R3 R4] are shown below:

*seq*(1) = 2*SR2 + 3*SR3 + SR4; [*seq*(1) SR1 SR2 SR3];

*seq*(2) = 2*SR3 +3*SR2 + SR3; [*seq*(2) *seq*(1) SR1 SR2]; and

*seq*(3) = 2**seq*(1) + 3*SR1 + SR2; [seq(3) *seq*(2) *seq*(1) SR1].

The n-state sequence detector corresponding to FIG. 10 is shown in FIG. 11. Because the sequence *seq* is shifted into the shift register, the same way as in the sequence generator, the output of the feedback loop is also the sequence *seq*. The input sequence *seq* and the generated sequence *seq* are both provided to a switching function *dt* that generates for instance symbols 0 when both inputs are identical. In fact '+' over *GF*(4) is such a function. When the symbols on the inputs of *dt* are not identical a symbol "not 0" will be provided.

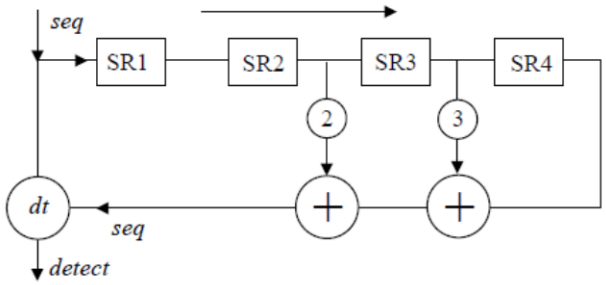


FIG. 11 A 4-state sequence detector

The shift register of the detector of FIG. 11 will be flushed if the initial contents of the shift registers of the generator and detector are not identical. Furthermore, when the structure of the detector of FIG. 11 does not correspond to the configuration of the generator of FIG. 10, then no sequence of all 0s will be generated by the detector. Because the detector is in a forward (rather than a feedback) configuration, it can start detection with any content of the shift register, for instance caused by a preceding sequence that is not generated by the sequence generator. The detector is flushed after 4 clock cycles, while the to be detected sequence can be much longer than 4 symbols. No continuous comparison of symbols is thus required to detect a generator generated sequence. [8].

*F. Synchronizing Sequence Generators*

Assume that a transmitter and a receiver both apply a shift register based 4-state sequence generator as illustrated in FIG. 12. The generator starts running with an initial shift register state [a b c d]. The function '+' is an addition over *GF*(4). The intermediate states at the outputs of the '+' functions are called *r*2 and *r*1, respectively. The inverter marked '2' is a multiplier 2 over *GF*(4). The state of '*out*' is *out*= 2**r*2 in terms of arithmetic over *GF*(4). The generated states of *out* at moments *t*1, *t*2, *t*3 and *t*4 are:

(1) at *t*1: $r1 = c+d$; $r2 = a+r1 = a+c+d$; and
*out* = 2*($a+c+d$); and
shift register: [2*($a+c+d$) *a b c*];
(2) at *t*2: $r1= b+c$; $r2 = 2a+b+3c+2d$; and
*out* = $3a+2b+c+3d$;
shift register:[($3a+2b+c+2d$) ($2a+2c+2d$) *a b*].
(3) at *t*3: $r1 = a+b$; $r2 = 2a+3b+c+2d$; and
*out* = $3a+b+2c+3d$; and
shift register: [($3a+b+2c+3d$) ($3a+2b+c+2d$) ($2a+2c+2d$) *a*];
(4) at *t*4: $r1=3a+3b+c+2d$; $r2 = 2b+3c+2d$; and
*out* = $3b+c+3d$; and
shift register: [($3b+c+3d$) ($3a+b+2c+3d$) ($3a+2b+c+2d$) ($2a+2c+2d$)].

One can continue the above calculations until *t*256, at which time the shift register content will return to [*a b c d*]. Remember to use the correct switching tables for arithmetic over *GF*(4), as these are different from mod-4 tables.

The initial content of the shift register may be considered as being 4 independent (and orthogonal) variables being expressed as *a*=[1 0 0 0]; *b*=[0 1 0 0]; *c*=[0 0 1 0]; and *d*=[0 0 0 1]. All output states as well as all shift register states depend only on these initial values.

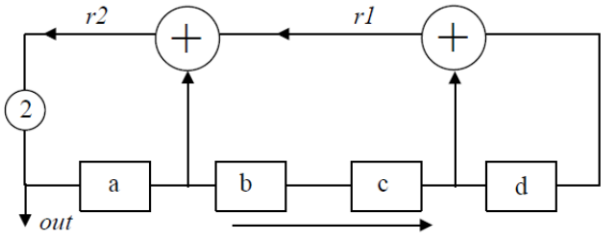

FIG. 12 A 4-state sequence generator

TABLE II.

| | | | | |
|---|---|---|---|---|
| 1000 | 0100 | 0010 | 0001 | 0 |
| 2022 | 1000 | 0100 | 0010 | 1 |
| 3213 | 2022 | 1000 | 0100 | 2 |
| 3121 | 3213 | 2022 | 1000 | 3 |
| 0201 | 3121 | 3213 | 2022 | 4 |

All additions and multiplications take place in accordance with *GF*(4) arithmetic. For instance a + b= [1 1 0 0] etc. Table II provides the consecutive 5 contents of the shift register as a function of the initial content. The content of the first shift register element is also the generated output state. This allows the calculation of the initial (or a current) shift register if 4 received symbols can be assigned to a predetermined generating moment. For instance, 4 synchronization symbols can be included in a received frame of symbols, or each frame has a specific sync symbol.

Assume that 4 consecutive symbols *s*1, *s*2, *s*3 and *s*4 of a sequence generated by the generator of FIG. 12 are transmitted in a frame for synchronization purposes. A starting state [a b c d] of a sequence generator can then be calculated, by knowing that s1= [1 0 0 0] = *a*; *s*2 = [2 0 2 2] = $2a + 0 +2c+2d$; *s*3 = [3 2 1 3]= $3a+2b+c+3d$, etc. By solving these equations over *GF*(4) one finds: *a*=*s*1; *b*=*s*1+*s*3+3*s*4; *c*=*s*1+*s*2+2*s*3+3*s*4; and *d*=2*s*2+2*s*3+3*s*4. Because it is known what the shift register content is relative to the received symbols, the shift register in the receiver can be provided with any required starting position. [9]

*G. Reversing the Flow*

Reed-Solomon (RS) coders are well known error-correcting block codes consisting of codewords with data symbols and check symbols. The check symbols are generally generated by a feedback shift register that receives data symbols and applies the final content of the shift register as the check symbols. In a decoder the data symbols in a received codeword are again applied to generate check symbols. If calculated check symbols are different from the received check symbols an error has occurred in the codeword. In order to correct the symbol in error, it has first to be determined which symbol(s) are in error. This is a relatively time consuming process in the known methods of error location in RS codes. In a novel approach, error positions can be found by evaluating intermediate comparative contents of the shifts register by running the coder first from the initial state (usually all 0s) to the end state (the check symbols) and

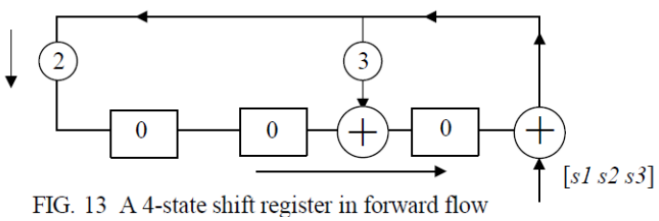

FIG. 13 A 4-state shift register in forward flow

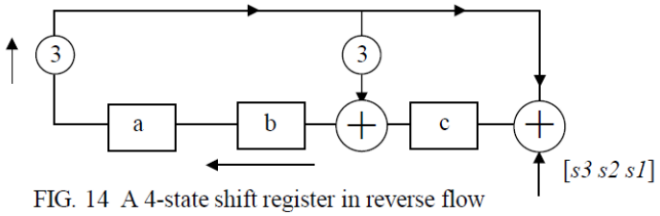

FIG. 14 A 4-state shift register in reverse flow

from the received end state (the received check symbols) back to the initial state, which should be all 0s. This process is explained in detail in [10]. Error location takes place by comparing intermediate shift register states in forward and reverse direction. Again, 4-state switching functions defined by *GF*(4) will be used. Due to required word distances RS codes usually require at least 8-state switching functions to adequately apply error correction, while real-life RS codes use 256-state switching.

FIG. 13 shows a 4-state shift register coder in Galois configuration with a signal flow in forward direction. The '+' is the addition over *GF*(4) and the inverters are multipliers over *GF*(4). Inputting the 4-state symbols *s*1, *s*2 and *s*3 into the coder with initial state [0 0 0] will provide end state of the shift register with: [*a b c*] = [(3*s*1+*s*2+2*s*3) (*s*1+2*s*2) (3*s*1+2*s*2+3*s*3). FIG. 14 shows the shift register operated in a reverse flow starting with the register content [*a b c*] and by entering the symbols *s*3, *s*2 and *s*1 in the reverse order of FIG. 13.

In reverse flow, the multiplier 2 of FIG. 13 is traversed in a reverse direction. The inverse of 2 is 3 (as 2*3=1 in *GF*(4)). The functions '+' are also traversed in a reverse direction. The function '+' in *GF*(4) is self reversing and thus remains the same in reverse direction. The original multiplier 3 is traversed in the same direction as in FIG. 13 and thus remains 3. Starting with the content [*a b c*], the end state of the shift register in reverse direction is [(3*a*+2*b*+*s*3) (3*a*+3*b*+2*c*+*s*2) (*a*+3*c*+*s*1)], which by proper substitution yields [0 0 0].

### *H. N-State Switching Notation*

The applied notation for expressing *n*-state switching is not consistently applied herein. All shift register applications as explained herein have been implemented and tested in Matlab®. [11] An n-switching table in Matlab is an array with dimension variables, such as *sc*4(*i*1,*i*2). For that reason, the array notation is used. For ease of understanding the recursive steps of the applications, it is much easier to apply an operator type notation, using '+' and '*', as defined over *GF*(*n*). For that reason, operator based notation is also used. However, these operators in Matlab are implemented as arrays.

## IV. NONBINARY SWITCHING

New physical switching technologies and materials are currently being developed, such as quantum computing and graphene, which inherently have nonbinary switching capabilities. Additionally, the consumption of data continues to grow rapidly, which puts tremendous pressure on the capacity for transmission, storage and processing of data. It is well known that transmission and storage of data in nonbinary form is more efficient than in binary form. Nonbinary transmission coding is well known. Storage of data in nonbinary, multi-level, form is currently a rapidly growing storage application.

The actual processing of nonbinary data with multi-valued circuitry is a desirable technology that will diminish the complexity of processing nonbinary symbols and may prevent the need to synchronize bit streams as series of words or as coded nonbinary symbols.

Currently, most data, even coded in binary form, is nonbinary data, either as characters coded as bytes, which represents a 256-level code, as pixels, which may be 24-bits, or as sound samples. Operations such as filtering, encryption and scrambling are currently performed in binary form. Intuition tells us that nonbinary processing of nonbinary signals or symbols should be easier and faster.

The nonbinary processing of nonbinary signals or symbols, even if coded or performed in binary form, has significant advantages over strictly binary presentations. A general impression is that nonbinary processing is in many ways an extension or similar to binary processing. Fortunately, that is not the case, as nonbinary switching functions offer much greater variety than binary switching functions. This aspect is not obvious to quite a number of developers, who are used to working with finite field applications over GF($2^p$). A finite field over GF($2^p$) is an extension field over GF(2) and in many ways and per definition looks like binary. The examples, provided herein and elsewhere in the Ternarylogic LLC portfolio [14] demonstrate that MVL is absolutely not limited to what is done in binary logic or has to look like it. In fact, there is a whole new world out there that is waiting to be explored and applied.

The logic design of nonbinary switching devices requires an implementation approach that can be automated, simulated, emulated and translated into a hardware implementation. Behavior of nonbinary switching devices is defined by the switching table or switching vector (in case of an inverter) and can be implemented on a processor by a nonbinary array. Combined with A/D and D/A converters, such an implementation behaves as a nonbinary device. Which nonbinary array to select is determined by its application. This is demonstrated in the previous pages by analyzing properties and desired properties of shift register based devices.

Shift register based applications and circuits are among the most widely used applications in data communication. The use of nonbinary elements and switching elements is not well known and offers novel ways to apply and/or analyze MVL functions and sequences. Analysis of binary and nonbinary shift register based sequences commonly takes place in terms of polynomial arithmetic over *GF*(n) rather than in terms of nonbinary switching functions. Furthermore, it is generally assumed that functions over *GF*(n), especially the addition over *GF*(n), are realized with XOR functions. This tutorial

provides a broader approach that allows the use of any useful n-state switching function, including n-state inverters.

The approach explained herein, can be successfully applied to create new devices, even if realized in binary components. Examples are a new type of wireless lock [12] and reversible annoyance masks for video protection. [13]

The shift register applications herein are relatively easy to visualize, because of the flow through the register elements, the switching functions and inverters.

There continues to be a significant interest in shift register based devices and implementations, especially related to encryption, such as non-linear aspects of shift registers with feedback (NLFSRs). [15] Not surprisingly, much of the effort is focused on the binary aspects of NLFSRs. One can easily imagine that advantages of binary NLFSRs will be even stronger in nonbinary NLFSRs. Shift register based devices form just one class of MVL implementations that needs addressing. Other MVL implementations are for instance directed to machine arithmetic and related devices such as digital filters and applications in memory and storage.

Nonbinary implementations or logic designs will be required for 100s, if not 1000s of basic designs that are now commonly performed in binary logic, ranging from memory devices, arithmetic, encryption, control functions and many others that may not lend themselves to the easy visualization such as LFSRs.

## V. MVL INVENTION FACTORY

Computers are now so common that many people tend to forget or even ignore that a computer is merely a machine.

When MVL becomes main stream there will be a huge demand for MVL logic designs. The fact that these designs can be directly realized in a novel nonbinary technology will be of little help. For MVL to be a successful technology, it needs both the nonbinary material/switching techniques and the nonbinary logic designs.

We are in MVL in a stage somewhat comparable to Claude Shannon when he wrote his Master Thesis at MIT. [16] At that time the binary switches were available to him, but not the design approach which he invented. In MVL we know the type of switches we want, but we do not have the MVL designs that are needed.

Some quick study and searching will easily generate 1000s of binary logic designs that need to be "MVL ported." For instance, a search in the USPTO Patent database, on the terms binary and logic in patent claims, will generate over 8000 patents. A search for the term digital in claims will generate over 200,000 patents. The listed titles of these patents alone will indicate where new inventions in nonbinary machine logic will be required. Currently, there is no automatic port from binary to nonbinary, with some exceptions, that will take full advantage of MVL.

Once, MVL is well established, it is predictable that there will be an effort to designate any patent of a "ported" binary-to-nonbinary invention as "obvious." However, at this time, as of 2015, there is no well-established nonbinary implementation methodology. It is not clear which nonbinary n-state technology will become dominant at what value of n.

Will we start with small increases of n? Such as n=3 or n=4? Or will we make a jump to large values of n, like n=32 or n=256? These uncertainties make most of the new nonbinary inventions, when claimed as using nonbinary switches, novel and non-obvious, it is believed.

Ternarylogic considers creating an MVL Invention Factory to take advantage of the need for future designs. It is believed that we can generate rapidly at least 150 additional inventions that are novel, non-obvious and patentable and that are not in the field of LFSRs.

You can learn more about us on www.ternarylogic.com. The site refers to a package, which is not cheap, that will provide additional background to our technology. You may also contact us at admin@ternarylogic.com.

## REFERENCES


[1] Gerrit A. Blaauw, "Digital system implementation," Prentice-Hall, Englewood Cliffs, N.J., 1976.

[2] Gerrit A. Blaauw, Frederick P. Brooks, "Computer architecture: concepts and evolution,"Addison-Wesley, March 1997.

[3] Peter Lablans, "Multi-valued digital information retaining elements and memory devices," U.S. Patent Ser. No. 7,397,690, issued on July 8, 2008.

[4] Peter Lablans, "Methods and systems for determining characteristics of a sequence of n-state symbols," U.S. Patent App. Pub. Ser. No. 20140032623, published on Jan. 30, 2014.

[5] Renato Fracassi and Tarmo Tammaru, "Data scrambler," U.S. Patent Ser. No. 4,304,962, issued on Dec. 8, 1981.

[6] Peter Lablans, "Binary and n-valued LFSR and LFCSR based scramblers, descramblers, sequence generators and detectors in Galois configuration," U.S. Patent Ser. No. 7,487,194, issued on Feb. 3, 2009.

[7] Peter Lablans, "Error correcting decoding for convolutional and recursive systematic convolutional encoded sequences," U.S. Patent Ser. No. 7,877,670, issued on Jan..25, 2011.

[8] Peter Lablans, "Multi-valued scrambling and descrambling of digital data on optical disks and other storage media," U.S. Patent Ser. No. 8,225,147, issued on July 17, 2012.

[9] Peter Lablans, "Method and apparatus for rapid synchronization of shift register related symbol sequences," U.S. Patent Ser. No. 8,817,928, issued on Aug. 26, 2014.

[10] Peter Lablans, "Methods and systems for rapid error correction by forward and reverse determination of coding states," U.S. Patent Ser. No. 8,645,803, issued on Feb. 4, 2014.

[11] Peter Lablans, "The Logic of More - Applications in Nonbinary Machine Logic," Published by Ternarylogic LLC, Morristown, NJ, 2014.

[12] Peter Lablans, "Method and apparatus for rapid synchronization of shift register related symbol sequences," U.S. Patent Ser. No. 8,817,928, issued on Aug. 26, 2014.

[13] Peter Lablans, "Methods and systems for rapid error correction by forward and reverse determination of coding states," U.S. Patent Ser. No. 8,645,803, issued on Feb. 4, 2014.

[14] Portfolio of Ternarylogic LLC published inventions, www.ternarylogic.com/portfolio.pdf.

[15] A List of Maximum Period NLFSRs, Elena Dubrova, Royal Institute of Technology (KTH), March 2012, http://eprint.iacr.org/2012/166.pdf.

[16] A Symbolic Analysis of Relay and Switching Circuits, Claude Elwood Shannon, 1937, at http://dspace.mit.edu/handle/1721.1/11173.