MAXIMAL SEQUENCES FROM NON-LINEAR SHIFT REGISTERS

by

John G. McEntyre, B.A.Sc.

Submitted in partial fulfillment of the requirements
for the degree of Master of Science.

Department of Electrical Engineering
Faculty of Pure and Applied Science
The University of Ottawa
Ottawa, CANADA

1968

ABSTRACT

The problem considered in this work is to develop a programmable algorithm which will produce the set of internal non-linear logics for feedback shift registers, which generate maximal (de Bruijn) sequences. Three cases are considered: (a) generation of one de Bruijn sequence (and its reciprocal), (b) generation of a few ($\leq 2^{n-1}$) sequences, and (c) generation of all sequences. For each case, a computer program has been written to generate the sequences and the corresponding feedback shift register logics. A further contribution is the development of a translation scheme which converts the inclusive-or sum of products form of a Boolean function to the exclusive-or sum of products and vice-versa.

Shift registers and their sequences are introduced with a survey of the basic linear theory leading up to the non-linear case. The solution to the problem presented is divided into two parts: the first part deals with some methods which will generate the actual maximal sequences desired; the second part is the realization of the circuit which will generate the desired sequence. The computer programs are briefly discussed as to their manner of approach and methods of use; they are block diagrammed and the source programs are listed.

ACKNOWLEDGMENTS

It was Dr. Janusz Brzozowski, formerly of the Department of Electrical Engineering (now at the University of Waterloo) who accepted the role of guidance throughout this dissertation. For this invaluable contribution, implemented throughout innumerable sessions of discussion and criticism, the author gratefully recognizes his patient stimulation which made this work possible.

Dr. Wayne A. Davis of the Defense Research Telecommunications Establishment, was responsible for introducing the author to related problems which led to the choice of this thesis topic. The author wishes to convey his sincere thanks for this, as for his continual encouragement.

The financial support of the Northern Electric Research and Development Laboratories of Ottawa, Canada, is gratefully acknowledged.

TABLE OF CONTENTS

1. INTRODUCTION

Binary sequential circuits have, within recent years, been studied in great detail. In particular, the constant input or autonomous case has been widely used in sequence generation with many and diverse applications: in communication systems, in digital ranging and tracking systems, and in digital computer and control systems.

The feedback shift register (FSR) is a simple structure of the class of autonomous sequential circuits and provides economical and efficient advantages as a standard design. These circuits, because of their ability to generate maximal sequences, enjoy various applications in efficient and reliable communications, when used for code generation and sequence recognition or decoding and as error-correcting circuits. Due to the auto-correlation properties of the generated sequences, they can be employed in high background noise situations, in such cases as secure communications or in radar ranging systems. Other related applications include unconventional radar, range measurement systems and correlation guidance systems.

Distinct advantages in circuit economies over conventional counter designs emphasize the FSR's importance in an extensive variety of situations arising in electronic and digital computer and control work as for sequencing and timing schemes.

Furthermore, the pseudo-random properties of the maximal sequences make them ideal for application in deterministic simulation of random processes as in Monte Carlo statistical techniques or as a model for the study of Markov processes or of finite state machines.

The design of shift registers using linear feedback functions is a well established technique and has been used for the generation of maximal sequences. However, removal of the restriction that the feedback logic be linear, greatly increases the number of maximal length

sequences available, from less than $2^n/n$ to exactly $2^{2^{n-1}}/2^n$, where an n-stage FSR, is an FSR of degree n when it generates the maximal period sequence of degree n. This is an astronomical increase and motivates the quest for methods of design of non-linear FSR's.

A further advantage of the non-linear sequences generated by such FSR's is evident in cryptography: a linear feedback function can be readily determined with the knowledge of only 2n consecutive bits of the sequence as such sequences are equivalent to delay-and-add sequences; in contrast, the non-linear feedback function requires 2^{n-1} consecutive bits to be determined.

While there has been a substantial number of studies made on the design of linear FSR's, no algorithm has ever been presented which would yield all or mostof the non-linear feedback logics required to generate maximal sequences. The present work was undertaken with this goal, in order to program the methods on a digital computer thereby also acquiring standard (since there is determinacy) pseudo-random sequences.

It was decided not to approach the problem from a polynomial behaviour standpoint which had proved quite successful with the linear case. This was settled after realizing that, after extensive studies, Golomb[12] was able to conclude that: "Sufficient data were obtained to establish conclusively the absence of a simple algebraic theory governing the sequences obtained from the various non-linear logics".

2. SURVEY

2.1 Shift Register Circuits

Definition 2.1.1: A sequential circuit[3] is a structure consisting of s inputs ($x_1, x_2, \ldots x_s$), r outputs ($z_1, z_2, \ldots z_r$), n unit delays with outputs ($y_1, y_2, \ldots y_n$), and (n + r) combinational circuits ($f_1, f_2, \ldots f_n, g_1, g_2 \ldots g_r$) such that: the next state variables are

$$y_i' = f_i (y_1, y_2, \ldots y_n, x_1, x_2, \ldots x_s)$$

$$\text{for } i = 1, 2, \ldots n \qquad (2.1.1)$$

and the present output is

$$z_j = g_j (y_1, y_2, \ldots y_n, x_1, x_2, \ldots x_s)$$

$$\text{for } j = 1, 2, \ldots r \qquad (2.1.2)$$

The combinational functions are Boolean, and for binary sequential circuits, the inputs, outputs, and delay signals are binary.

A constant input (CI) circuit is one in which the input s-tuple is a fixed s-tuple of 0's and 1's.

If the next state is fully determined by equation 2.1.1, the machine is fully-specified[4] and there is a unique successor state for each state.

The set of output values of the unit delays, at one particular moment of time is a binary vector, and is the internal state Y of the binary circuit at that time. The binary vectors representing the next state Y' and the present output Z can be similarly and respectively defined as the subsequent set of output values of the unit delays and the present set of circuit outputs.

The behaviour of a sequential circuit can be characterized by

the state table. This is the straightforward listing of the next state (and present output) for each present state as shown in fig. 2.1.2 for the case of a CI circuit.

Present State	Next State	Present Output
Y_1	Y_1'	Z_1
Y_2	Y_2'	Z_2
⋮	⋮	⋮
Y_t	Y_t'	Z_t

Fig. 2.1.2 State Table

Each Y_k, Y_k', and Z_k from fig. 2.1.2 represents the vectors $(y_1, y_2, \ldots y_n)$, $(y_1', y_2', \ldots y_n')$, and $(z_1, z_2, \ldots z_r)$ respectively and at one particular moment of time as $k = 1, 2, \ldots t$. Thus, in the case of a complete state table, every state vector appearing in the next state column will also appear in the present state column; in this manner every state transition will be defined.

If each state has a unique previous state, the machine is called nonsingular, and each state will appear only once in the next state column of the state table.

The state graph is a convenient way of representing the state transitions pictorially as shown in fig. 2.1.3. If the machine is cyclic, all its states occur in closed cycles, hence, the state graph will appear as a set of closed cycles as in fig. 2.1.3.

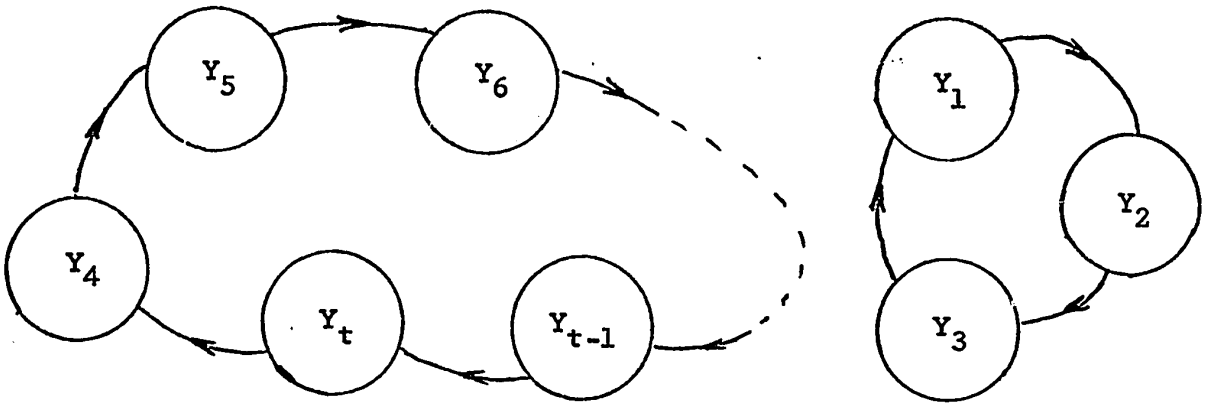


Fig. 2.1.3. State Graph (two cycles)

Definition 2.1.2: A feedback shift register (FSR) is a sequential circuit with combinational logic only in its single feedback loop.

We shall consider only CI FSR's which will be fully-specified and nonsingular.

Thus the FSR is a structure with the following state equations:

$$y_1' = f\ (y_1, y_2, \ldots y_n) \qquad (2.1.3)$$

$$y_i' = y_{i-1} \quad \text{for } i = 2, 3, \ldots n \qquad (2.1.4)$$

An FSR with n delay units is an n-stage FSR as shown in fig. 2.1.4.

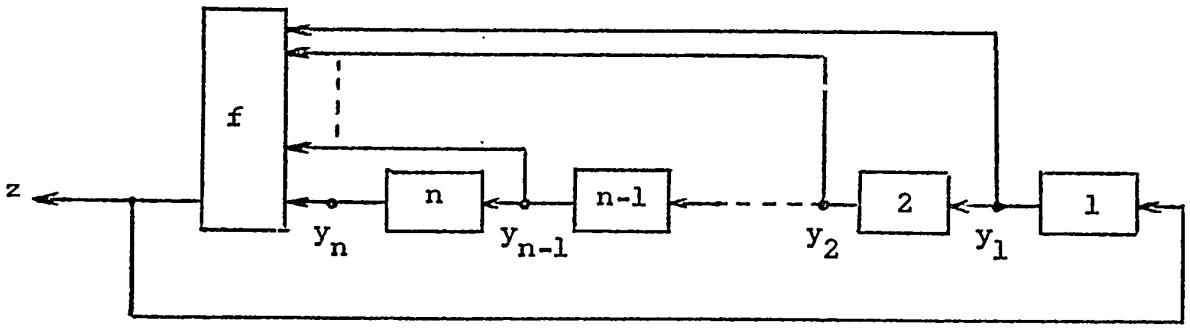


Fig. 2.1.4 n-stage FSR.

The function f in equation 2.1.3 is a logical Boolean function determined by the combinational networks in the feedback path, and is called the feedback function. In this work, the FSR output, z, is chosen to coincide with the feedback function as shown in fig. 2.1.4.

Some authors have defined their output as the output of the last delay, y_n, which defines the same output z, shifted through n units of time.

A linear (non-linear) FSR is one in which the feedback function is linear (non-linear).

In the algebra of linear functions, only two operations are defined : scalar multiplication ($\cdot$), and "exclusive - or" addition ($\oplus$) (the latter is also recognized as addition modulo 2).

Thus linear feedback functions are of the type:

$$f = y_1' = c_1 y_1 \oplus c_2 y_2 \oplus \ldots \oplus c_n y_n \oplus b \qquad (2.1.5)$$

The internal state of an FSR with such a feedback function can be expressed in matrix form:

$$\begin{bmatrix} y_1' \\ y_2' \\ \vdots \\ y_n' \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 & \cdots & & & c_n \\ 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & & & \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \oplus \begin{bmatrix} b \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad (2.1.6)$$

$$\text{i.e.} \quad Y' = T\,Y + B \qquad (2.1.7)$$

where T is called the Transition Matrix.

In the non-linear case, all feedback functions can be represented by the general sum of primary products[1]:

$$y_1' = a_0 (\bar{y}_1 \bar{y}_2 \ldots \bar{y}_n) + a_1 (y_1 \bar{y}_2 \ldots \bar{y}_n) +$$

$$+ a_2 (\bar{y}_1 y_2 \bar{y}_3 \ldots \bar{y}_n) + \ldots + a_{(2^n-1)} (y_1 y_2 \ldots y_n) \qquad (2.1.8)$$

where $a_i \in \{0, 1\}$, $i = 0, 1, \ldots 2^n-1$

and $\bar{y}_j$ is the complement of y_j.

This form is called the first canonical form (FCAN).

This same feedback function may be represented as an exclusive - or sum of all the possible non-complemented products of the secondary variables:

$$y_1' = b_o \oplus b_1 y_1 \oplus b_2 y_2 \oplus b_3 y_1 y_2 \oplus b_4 y_3 \oplus \ \ldots$$

$$\ldots \oplus b_{(2^n-1)} (y_1 y_2 \ \ldots \ y_n) \qquad (2.1.9)$$

where $b_i \in \{0, 1\}$, $i = 0, 1, \ldots 2^n-1$

This form will be called the exclusive - or canonical form (EXOR).

These forms are made up by the summing of a set of products, of the secondary variables, whose presence or absence is indicated by the binary parameters a_i or b_i preceding these products (see equations 2.1.8-9). As the set of products is general, individual feedback functions could be represented by the set of binary variables only, as long as the order of the products is standardized. The order, as given by equations 2.1.8-9, has been chosen as it allows a bijective mapping of the ordered position of the binary variable as given by its subscript, onto the particular product functions of the secondary variables. (Furthermore, this critical ordering is a definite prerequisite for section 3.7.)

The ordering method that incorporates the bijective mapping is a binary ordering, considering only the non-complemented secondary variables: the method considers the presence of y_j to be the presence of a 1 bit in the j^{th} position of the binary number representing the binary variable subscript; and conversely, each 1 bit in the binary variable subscript (in the i of a_i or b_i), when such subscript is interpreted in binary form, represents the presence of the corresponding non-complemented secondary variable. As the EXOR consists only of non-complemented variables, this result is complete; in the FCAN case, as

each product consists of all secondary variables, the missing ones (if any) are to be included in complemented form. This method is carried out to include the 0 subscript with no binary bits in the subscript, hence it represents no non-complemented variables: in the FCAN case it represents the all complemented variable case $(\bar{y}_1\bar{y}_2 \cdots \bar{y}_n)$, and in the EXOR case, it represents the function 1 to be summed with the products.

Example 2.1.1

Consider the binary parameter with subscripts 0, 4, 5, and 7; i.e. a_o or b_o, a_4 or b_4, a_5 or b_5, and a_7 or b_7. The corresponding secondary variable product function is developed as follows:

Subscript	Position for			Secondary variables if	
i	y_3	y_2	y_1	b_i	a_i
$0 = (0$	0	$0)_2$		1	$\bar{y}_3\,\bar{y}_2\,\bar{y}_1$
$4 = (1$	0	$0)_2$		y_3	$y_3\,\bar{y}_2\,\bar{y}_1$
$5 = (1$	0	$1)_2$		$y_3\,y_1$	$y_3\,\bar{y}_2\,y_1$
$7 = (1$	1	$1)_2$		$y_3\,y_2\,y_1$	$y_3\,y_2\,y_1$

Thus the correspondence, if n = 3, is:

a_o	$\bar{y}_1\bar{y}_2\bar{y}_3$
b_o	1
a_4	$\bar{y}_1\bar{y}_2y_3$
b_4	y_3
a_5	$y_1\bar{y}_2y_3$
b_5	y_1y_3
a_7	$y_1y_2y_3$
b_7	$y_1y_2y_3$

Because of the availability of this bijective mapping which standardizes the order of the products as defined in equations 2.1.8-9, it is now possible to express these equations as simple 2^n- tuples of the binary variables (a_i and b_i). Thus the FCAN can be expressed as

$$(a_o a_1 \cdots a_{2^n-1}) \qquad (2.1.10)$$

and the EXOR form as

$$(b_o b_1 \cdots b_{2^n-1}) \qquad (2.1.11)$$

Example 2.1.2

Consider the 2^3- tuple 10001101. It consists of four 1 bits: one each in the 0^{th}, 4^{th}, 5^{th}, and 7^{th} position when enumerating from left to right as indicated in equations 2.1.10-11.

Interpreting these four 1 bits yields the four secondary variable products either as a_o, a_4, a_5, and a_7 or as b_o, b_4, b_5, and b_7 as was shown in example 2.1.1.

If the 2^n- tuple is an FCAN, the feedback function would be:

$$y_1' = \bar{y}_1\bar{y}_2\bar{y}_3 + \bar{y}_1\bar{y}_2 y_3 + y_1\bar{y}_2 y_3 + y_1 y_2 y_3 \qquad (2.1.12)$$

And if the 2^n- tuple is an EXOR:

$$y_1' = 1 \oplus y_3 \oplus y_1 y_3 \oplus y_1 y_2 y_3 \qquad (2.1.13)$$

(NOTE: Equations 2.1.12 and 2.1.13 do not represent the same function. This indicates that the 2^n- tuple representations may differ for the two canonical forms. A method for converting these canonical forms is presented in section 3.7.)

The truth table of an FSR is the listing of the new or output bit generated for each internal state as shown in figure 2.1.5.

Internal State				Output
y_n	y_{n-1}	y_2	y_1	y_1'
0	0	0	0	b
0	0	0	1	$b \oplus c_1$
⋮	⋮	⋮	⋮	⋮
1	1	1	1	$b \oplus \sum_{i=1}^{n} c_i$

Fig. 2.1.5 Truth table of linear FSR

Figure 2.1.5 shows the output of the linear case, compare to equation 2.1.6; for the more general case, the output column would contain the binary bits as found from the feedback function equation 2.1.3. The truth table represents the FSR's state behaviour as the feedback function affects only the first delay element. In this way it is actually a more simplified state table due to the FSR's more simple state equations (equations 2.1.3-4).

Thus, the next state vector is now given from the previous one by replacing only one bit:

$$Y_i = (y_1, y_2, y_3, \cdots, y_n) \qquad (2.1.14)$$

$$Y_i' = (y_1', y_1, y_2, \cdots, y_{n-1}) \qquad (2.1.15)$$

where $i = 1, 2, \ldots, 2^n$.

Every FSR determines a unique output column in its truth table format. If the internal state column is ordered in the critical 2^n- tuple canonical form (as in equation 2.1.8), the output column is actually the FCAN-tuple. If the internal state column is ordered as the states appear in the FSR cycle, the output column will list

each output in proper sequence; this will constitute the FSR's output sequence. As each FSR determines a unique output column, it determines a unique output sequence which will be considered in the next section.

2.2 Sequences

The set of output bits of the FSR as the machine cycles through its internal states, constitutes the output sequence of the FSR. As the logic of the FSR is binary, the generated sequence is a binary sequence.

Obviously, if the FSR is cyclic, its sequence is periodic with periodicity equal to the number of states in the state cycle.

A sequence is[12] recursive of degree n, if each set of n consecutive bits, (n-bit word) occurs at most once in one period. At one unit of time, the past n output bits define the state vector; hence each n-bit word is a state of the cyclic FSR's state graph. As all states in one state cycle are unique for a nonsingular FSR, the n-bit word states, in the state cycle, are unique; furthermore, the length of these n-bit words is equal to the number of stages of the FSR; hence, the output sequence will be recursive of degree equal to the number of stages of the FSR and with period equal to the length of the cycle.

Random sequences have certain properties: the balance property, the run property and the auto-correlation property. If a deterministic sequence also has these properties[11], it is termed a pseudo-random sequence. For binary sequences of 0's and 1's, a sequence with an equal number of 0's and 1's in any period is balanced. Having the

run property means that in every period, half the runs have length one, one fourth have length two, one eighth have length three, which is to say that $1/(2^n)$ have length n; also, these runs are all balanced. In other words, there are for both the 0 and the 1 case, two runs of length k for each run of length (k + 1).

A maximal period sequence of degree n has these pseudo-random characteristics, and is recursive of degree n with period 2^n. This is the non-linear case, known as a de Bruijn sequence; in the special linear case, the longest run of 0's has n-1 0's instead of n 0's, hence has period (2^n-1) and is termed an m-sequence.

Example 2.2.1

Consider the following sequences, S_i, and their respective properties:

S_1 = 00000001 00000001

S_2 = 111010 111010

S_3 = 110100 110100

S_4 = 00111001 00111001

S_5 = 11101000 11101000

S_1 has period 8 but is not recursive of any degree <7 and has no run properties.

S_2 is recursive of degree 3 yet does not have period 2^3, it has period 6; also it is not balanced as there are twice as many 1's as there are 0's.

S_3 is balanced and does have run properties, yet while it is recursive of degree n = 3, it does not have a period of 2^n, its period is 6.

S_4 is balanced and has a period of $2^3 = 8$, yet it is not recursive of degree 3 as the n-bit word 001 occurs twice in one period.

This sequence also does not have run properties.

S_5 is recursive of degree n = 3 and has period $2^n = 8$; it also has run properties and is thus a maximal period sequence.

2.3 Historical Background

The theory governing sequences was originally spurred on by dealings with reoccurring decimals. This led to a more general approach to sequences and then to sequence generators, of which the FSR's are an important group. Some of these earlier findings are of historical interest as well as of theoretical significance.

Good[15] inspired by Borel's proof that almost all decimals are periodic, proved the existence of a binary sequence of periodicity 2^n and which was recursive of degree n, for any n. His proof was based on Euler's unicursal theorem, and used a topological approach to a set of diagrams (now known as Good's Diagrams) consisting of a node for each different n-digit word. He thus proved the existence of a path through all nodes, thereby proving the existence of maximal sequences.

Rees[25] was able to prove a similar case using the theory of finite fields instead of Good's combinatorial construction; but in Rees' case, his sequence, while containing every word of n digits, lacks the one of n zeroes; this, as was seen in section 2.2 and will be explained subsequently, is the distinction between the linear and the non-linear case.

But it was de Bruijn[7] who, using a diagram similar to Good, and who apparently discovered it independently[12], was able to find the number of such maximal sequences. His proof, using these diagrams he called "T-nets", considered "complete walks" for each n,

then a "doubling" process to consider the case for (n + 1); considering N as the number of "walks" (maximal sequences) for degree n, he found and proved the number of walks of degree n + 1 (N*).

$$N^* = 2^{(2^{n-1}-1)} N \qquad (2.3.1)$$

In this way he was able to calculate the number of maximal sequences of degree n, (N_n) that exist.

$$N_n = 2^{(2^{n-1}-n)} \qquad (2.3.2)$$

In his memory, these maximal sequences are referred to as de Bruijn sequences.

As implied earlier, a maximal sequence with periodicity 2^n is termed non-linear; this is a direct consequence to the FSR which will generate such a sequence. This consequence is based on the recursive properties of the sequence. As the FSR shifts one bit at a time, generating one new bit per shift, it retains n bits at any one time, which is the state of the FSR; this is the reason for the importance of considering the n-bit word in the sequence. Further, the uniqueness of this appearance of an n-bit word is simply the uniqueness of the state in its appearance in one cycle.

In a linear FSR, the zero state must be succeeded by the zero state, as the state equation (2.1.6) must be zero as all the y_i's (i = 0, ..., 2^n-1) are zero. This zero state cycling to itself is considered a trivial cycle and therefore the maximum number of states possible that this FSR can cycle through is (2^n-1) states. Thus, the maximum sequence generated by such an FSR is of length (2^n-1), and is called an m-sequence.

Elspas[9], using parts of Galois field theory[1] of cyclotomic polynomials, calculated the number of m-sequences of degree n (M_n).

$$M_n = \varphi(2^n-1)/n \qquad (2.3.3)$$

where $\varphi(k)$ is the Euler phi (or Totient) function defined as the number of integers less than k which are relatively prime to k.

Thus, it is now evident that there exists a much greater number of non-linear maximal sequences than linear ones, which exposes one of the motives for an attempt to characterize them. Fig. 2.3.1 demonstrates this greater number.

	Sequences of degree n	
Degree n	m-sequences M_n (equation 2.3.3)	de Bruijn sequences N_n (equation 2.3.2)
2	1	1
3	2	2
4	2	16
5	6	2048
6	6	2^{26}
7	18	2^{57}
8	16	2^{120}
9	48	2^{247}
10	60	2^{502}

Fig. 2.3.1 Comparing de Bruijn's to Elspas' number.

2.4 Linear Theory

Much work has been done to characterize the linear sequential machine, and the linear feedback shift register, including the ones which generate m-sequences.

The main impulse[9] for the exhaustive work accomplished in the linear field is due to the fact that linear sequential machines consist entirely of three kinds of elements: unit-delay elements, modulo-p adders, and modulo-p scalar multipliers; these operations constitute linear operations over the modular field of integers. Thus linearity made such networks amenable to mathematical analysis.

In the case of p = 2 (which is of interest to us) multiplication by 0 and 1 corresponds to open and short circuits, respectively.

The network considered here consists of an FSR with no input and with a linear feedback function. Such a circuit is a linear FSR. Thus it is completely defined by its Boolean feedback function and the number of stages (n). The output of the circuit is chosen to be equal to the feedback function. If the output of a linear FSR consists of a sequence with period (2^n-1), the linear FSR will be called maximal as its output is a maximal period linear sequence (m-sequence).

As indicated earlier, it has been shown[11] that linear maximal FSR's exist for any n, and the number of such maximal FSR's is M_n (see equation 2.3.3) for a given n.

The problem of analysis is: given a linear FSR, does it generate an m-sequence? The brute force solution to this problem would be to construct the machine's state diagram and thus test whether the states cycle through 2^n-1 states. This approach, while effective, is tedious and does not give any indication of the general properties of the circuit.

The analysis of linear FSR's can be accomplished quite readily using certain characterizations of the FSR's state behaviour: the transition matrix, and its characteristic polynomial.

The transition matrix, as shown in equations 2.1.6 and 2.1.7,

represents the state equations, since linear FSR's have no input, (i.e. X = 0).

More particularly, given an n-stage linear FSR as shown in Fig. 2.4.1, with feedback function

$$f = c_1 y_1 \oplus c_2 y_2 \oplus \ldots \ldots \oplus c_n y_n \qquad (2.4.1)$$

the transition matrix is

$$T = \begin{bmatrix} c_1 & c_2 & c_3 & \ldots & c_{n-1} & c_n \\ 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ldots & 0 & 0 \\ 0 & 0 & 0 & \ldots & 1 & 0 \end{bmatrix} \qquad (2.4.2)$$

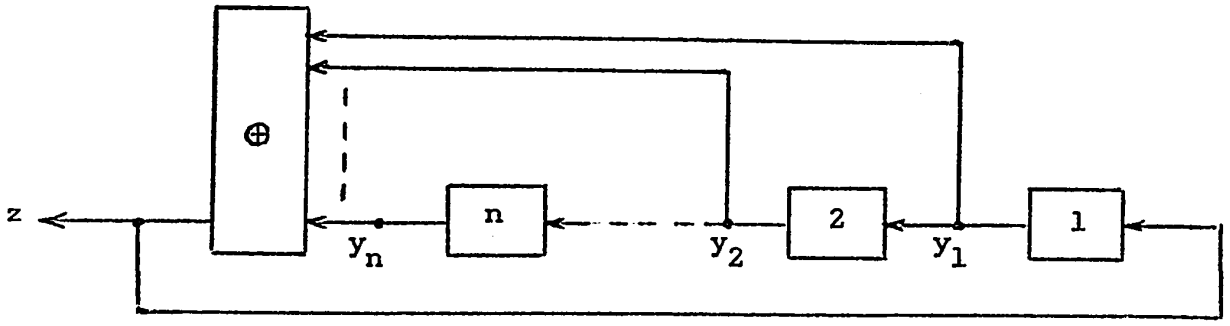


Fig. 2.4.1 n-stage linear FSR

Let Y denote the column state vector

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \qquad (2.4.3)$$

then the circuit is uniquely characterized by the equation:

$$Y' = T\,Y \qquad (2.4.4)$$

Using matrix theory[1], it is straightforward to test whether a

square matrix is nonsingular, i.e. it is if its determinant is different from zero:

$$|T| \neq 0 \qquad (2.4.5)$$

If so, the matrix has a unique inverse (T^{-1}) which corresponds to the machine having a unique predecessor (previous state) for each of its states. Such a machine is termed a nonsingular machine, and its state behaviour is cyclic. Thus for m-sequences we need only consider nonsingular matrices.

For any matrix M, there exists a unique characteristic polynomial[1] defined by $|M-xI|$, where I is the Identity matrix. Hence, the characteristic polynomial of the transition matrix is given by

$$F(x) = |T - xI| \qquad (2.4.6)$$

One verifies that

$$F(x) = x^n - c_1 x^{n-1} - c_2 x^{n-2} - \ldots - c_n \qquad (2.4.7)$$

Because of the special form of the transition matrix (equation 2.4.2), the matrix can be uniquely constructed from its characteristic polynomial.

Much information is available [9,11,22] about the linear FSR, and hence the generated m-sequence, from the algebraic properties of the polynomial. The largest period of the generated sequence, or the exponent of the characteristic polynomial of the transition matrix, is given[11] by p, when p is the smallest positive integer for which the polynomial, F(x), divides $1-x^p$.

This periodicity test can be accomplished directly using the transition matrix: since the Cayley-Hamilton theorem[1] states that every square matrix satisfies its own characteristic polynomial equation, then

$$F(T) = 0 \qquad (2.4.8)$$

Now if F(x) divides $1-x^p$, then T is a root of $I-T^p$; thus, simply find the smallest p such that

$$T^p = I \qquad (2.4.9)$$

If the polynomial is irreducible[9], all non-trivial periods of the linear FSR are of equal length. This implies that one linear FSR has many cycles; the FSR will be in a particular cycle depending on its starting state, and will continue in the cycle containing this starting state.

A linear FSR in which all states except the all zero one form a single cycle, is a maximal period linear FSR; this machine generates an m-sequence. The properties of its corresponding polynomial are well known: Peterson[22] organises the pertinent Galois field theory necessary for this study. Thus the synthesis of maximal period linear FSR or of m-sequence generators can be approached from the algebraic point of view.

Zierler[30] and Golomb[11] showed that a necessary condition for maximal length is that the polynomial be irreducible.

Sufficiency is established[9] if the polynomial is also primitive: i.e. it is not a divisor of x^k-1 for any integer $k<2^n-1$.

All k such that F(x) divides x^k-1, for $k\leq 2^n-1$, must be[22] divisors of 2^n-1. If 2^n-1 is a prime (then called a Mersenne prime) for some n, then the only possible k is $k = 2^n-1$. Hence, every irreducible polynomial of degree n such that 2^n-1 is a Mersenne prime, characterizes a maximal FSR.

The synthesis of m-sequences for any n is available from tables[21,22] of irreducible primitive polynomials for $n \leq 19$ and of Mersenne primes; for polynomials of greater degree, Peterson[22] outlines a search method which is applicable to computer programming.

In summary, the transition matrix:

1) Is a unique characterization of the linear FSR.
2) Can be used for testing the singularity of the linear FSR.
3) Can be used for testing the periodicity of the linear FSR.
4) Defines the characteristic polynomial.

The characteristic polynomial:

1) Is a unique characterization of the linear FSR.
2) Can be used for testing the periodicity of the linear FSR.
3) If irreducible, implies cycles of equal length.
4) If irreducible and primitive, implies a maximal period linear FSR.

Another polynomial approach for the analysis of the linear FSR, using the network directly, was introduced by Golomb[11], who called it the characteristic polynomial of the shift register, h(x).

$$h(x) = 1 + \sum_{i=1}^{n} c_i x^i \qquad (2.4.10)$$

This polynomial is closely related to the polynomial H(x) which is the time inverse of F(x) (equations 2.4.6-7).

$$H(x) = \frac{F(x)}{x^n}$$

$$= 1 + c_1 x^{-1} + c_2 x^{-2} + \ldots + c_n x^{-n} \qquad (2.4.11)$$

The only difference between H(x) and h(x) is the negative sign of the exponents of H(x) which can be interpreted[26] as having significance in the time domain only: negative exponents merely refer to preceding time intervals. Hence one polynomial can be considered the inverse or the reciprocal of the other.

As the reciprocal of an irreducible polynomial is an irreducible polynomial, and the reciprocal of a primitive polynomial is a primitive polynomial[22], the investigation of either the polynomial of the matrix, or of the polynomial from the FSR yields similar results.

Example 2.4.1

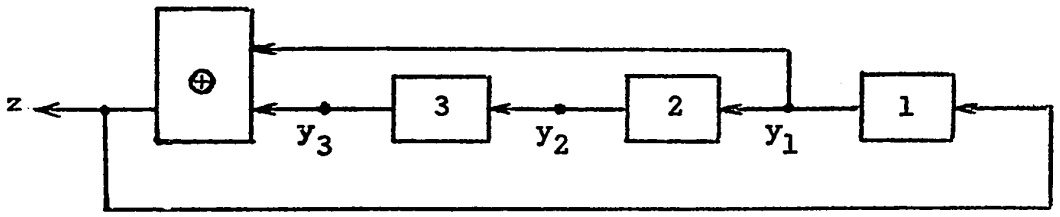


Fig. 2.4.2 3-stage linear FSR.

1) The internal state equations in matrix form are:

$$\begin{bmatrix} y_1' \\ y_2' \\ y_3' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

which is the matrix equation:

$$Y' = T\,Y$$

where T is the transition matrix.

2) The transition matrix is nonsingular as:

$$\begin{vmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} = 1 \neq 0$$

3) The period is p = 7 since

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^7 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I$$

4) The characteristic polynomial is:

$$F(x) = \begin{vmatrix} 1-x & 0 & 1 \\ 1 & -x & 0 \\ 0 & 1 & -x \end{vmatrix} = (1-x)(x^2) + 1 = x^3 + x^2 + 1$$

5) Alternatively, p = 7 because:

$F(x) = x^3 + x^2 + 1$ divides $1-x^7$

$(x^7 + 1)/(x^3 + x^2 + 1) = x^4 + x^3 + x^2 + 1$

6) F(x) is irreducible, as it has no factors, hence all non-trivial cycles are of length 7.

7) F(x) is primitive as the minimum k which allows F(x) to divide $1-x^k$ is $k = 7 = 2^n-1$.

8) The characteristic polynomial of the FSR

$$h(x) = 1-x-x^3 = 1+x+x^3$$

9) The time inverse of F(x) is

$$H(x) = {}^{F(x)}/_{x^n} = \frac{x^3+x^2+1}{x^3} = 1+x^{-1}+x^{-3}$$

which is the same as h(x), except for the negative exponents.

Young[29] lists several additional theorems for periods which are simply realizable with a single 2-input mod-2 adder (exclusive-or gate) feedback circuit.

Huffman's[16] methods of analysis are slightly different; his technique uses operator polynomials. He also discussed in considerable detail the realization of a network structure possessing the required characteristic polynomial.

The binary situation has been generalized[9,11] to that of multivalued p-nary logic, where p is any prime as suggested by Huffman[16]. (His work was restricted to the binary case.)

Further work was carried out by Scholefield[27] who was able to cut down the amount of feedback logic by using more than one FSR in series; and by Srinivasan[28], who extended the theory to include inputs.

2.5 Non-linear Developments:

Golomb[12] made a comprehensive study of the non-linear case which concludes with many useful results. By detailing a method of incorporating the all-zero state into the linear cycle, for any n, as suggested by Rees[25], Golomb proves the existence of a non-linear maximal cycle for any n. His study thus develops a method of fusing two non-maximal cycles together.

Magleby[20] did further work in cycle fusion, in order to merge all cycles into the one maximal cycle.

One disadvantage[12] of realizing non-linear maximal FSR's is the necessity of using all tap positions of the register, as all variables are included in the feedback function. This has important practical consequences. In contrast, a linear maximal FSR can be obtained with as few as two taps[29].

There were some attempts to modify the existing linear theory to include the non-linear case: Kautz, Elspas and Stone[18] increased the degree of the Characteristic Polynomial into their Associated Polynomial; Bryant and Killick[2] increased the degree of the Transition Matrix. The results did not prove very satisfactory.

Magleby[20] tried a more direct approach, by extending Edwards[8] theoretical methods and defining a new polynomial: the Describing Polynomial which yields the cycle lengths. This lack of success in algebraic approaches was prophesied by Golomb[12] who, after an exhaustive

study of certain low order polynomials, concluded: "sufficient data was obtained to establish conclusively the absence of a simple algebraic theory governing the sequence obtained from the various non-linear logics". Yet he did indicate that the feedback function of a cyclic FSR can be simplified by decomposition:[12]

$$y_1' = f(y_1, y_2, \cdots, y_n) \qquad (2.5.1)$$

$$= y_1' = g(y_1, y_2, \cdots, y_{n-1}) \oplus y_n \qquad (2.5.2)$$

Kautz[17] introduced certain design aids, while Kiyasu and Toda[19], implementing a search for all the non-linear maximal FSR's, were able to impose bounds on the weight of the feedback polynomial; this narrowed their search, yet still necessitated a test for acceptance or rejection.

The random properties of de Bruijn sequences was shown conclusively by Golomb[13] in his study using a statistical model to investigate the periodicity of FSR's. He was able to show that the output of an FSR follows the expected randomness of probability theory.

A more detailed discussion of some aspects of non-linear shift registers is given in the next section.

3. GENERATION OF MAXIMAL SEQUENCES BY NON-LINEAR FSR'S

3.1 Introduction

It is now evident that the linear case has been thoroughly studied and that there exist algorithms to realize the hardware necessary to generate m-sequences. It has been shown that methods are available to find the proper feedback connections of an FSR which will realize an m-sequence for any n, and how all such m-sequences can be realized for a given n.

Because of the advantages of de Bruijn sequences, it would be of

great use if similar algorithms were equally available for the non-linear case. This section proposes several such algorithms: two which yield some FSR realizations which generate de Bruijn sequences of any degree n, and one for the generations of all the de Bruijn sequences of any degree n.

It should be noted that hardware realizations of circuits generating de Bruijn sequences need not necessarily be FSR's. Actually, the more general sequential circuit can realize similar output sequences; furthermore, Scholefield[27] did show that, in some cases, multiple-register realizations can reduce the number of logic gates required. However, the scope of this work will be limited to FSR realizations due to the development trends towards modular hardware and thus the increasing reliability and efficiency in using such components.

3.2 FSR's with Output Logic

If a circuit generating a de Bruijn sequence of degree n is available, it can be shown that this circuit can generate all other de Bruijn sequences of the same degree through the use of output logic on the circuit configuration. In this light, the problem of generating all de Bruijn sequences is simplified to finding a single one of the proper degree. This basic machine is shown to be available by algorithms to be developed in section 3.3.

As the generator must cycle through 2^n states, the particular output logic to generate the particular bit of the de Bruijn sequence desired is directly available from the n bits of the state. Furthermore, as there are 2^n different cycling states and 2^n bits in the desired sequence, the output logic's complexity will depend on the starting state chosen. Thus, to find the most efficient realization would require a methodical check of all 2^n starting states prior to

making the choice.

Example 3.2.1:

Consider the de Bruijn sequence 01011100 with period 2^3. Assume that the following de Bruijn generator of the same degree is available:

de Bruijn generator							
Present State			Next State			Assigned Output	
y_3	y_2	y_1	y_3'	y_2'	y_1'	a	b
0	0	0	0	0	1	0	1
0	0	1	0	1	1	1	0
0	1	1	1	1	1	0	0
1	1	1	1	1	0	1	0
1	1	0	1	0	1	1	1
1	0	1	0	1	0	1	0
0	1	0	1	0	0	0	1
1	0	0	0	0	0	0	1

The realization of the generator outputting the required de Bruijn sequence is accomplished by assigning the proper output logic as indicated above in assigned output a. The required output logic would then be:

$$L_a = y_1\bar{y}_2\bar{y}_3 + y_1y_2y_3 + \bar{y}_1y_2y_3 + y_1\bar{y}_2y_3$$

This output logic can be of reduced complexity if the most efficient starting state was found and used. In this case starting with state 111 would be advantageous as indicated in the example as the assigned output b. In this case the required output logic would be:

$$L_b = 1 \oplus y_1$$

The drawback of requiring the use of extra logic coupled to the

necessity of testing for minimum component realization further indicated the desirability of developing a direct method which would be output logic free.

There exists[12] a set of sequences which share with de Bruijn sequences the same periodicity property and the same run properties, but because these sequences do not have the recursive properties, they cannot be generated by FSR's (without output logic) with a minimum number of delays. These run distribution sequences (RDS) are more numerous than de Bruijn sequences. The combined number of de Bruijn sequences and RDS's is given by[12]:

$$\frac{1}{2^{n-2}} \left\{ \frac{(2^{n-2})!}{\prod_{i=3}^{n} (2^{n-i})!} \right\}^2 \sim \frac{2^{2^n - \frac{(n-3)(n-2)}{2}}}{16C\,(2\pi)^{n-3}} \qquad (3.2.1)$$

where $e^2 < C < e^{2.33}$

In order to generate RDS's using minimum delay FSR's, the output logic approach is quite practical.

Example 3.2.2:

Consider the RDS 1111000010101100, which is not recursive, as the words 0101 and 1100 both occur twice, yet which has all the run distribution properties.

To implement this sequence, use the de Bruijn generator assumed to be available:

de Bruijn generator								
Present State				Next State				Assigned Output
y_4	y_3	y_2	y_1	y_4'	y_3'	y_2'	y_1'	
0	0	0	0	0	0	0	1	1
0	0	0	1	0	0	1	1	1
0	0	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	0	0
1	1	1	0	1	1	0	1	0
1	1	0	1	1	0	1	0	0
1	0	1	0	0	1	0	0	0
0	1	0	0	1	0	0	1	1
1	0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	1	1
0	1	0	1	1	0	1	1	0
1	0	1	1	0	1	1	0	1
0	1	1	0	1	1	0	0	1
1	1	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0

The required output logic function L would then be:

$$L = \bar{y}_1\bar{y}_2\bar{y}_3\bar{y}_4 + y_1\bar{y}_1\bar{y}_2\bar{y}_4 + y_1y_2\bar{y}_3\bar{y}_4 + y_1y_2y_3\bar{y}_4$$
$$+ \bar{y}_1\bar{y}_2y_3\bar{y}_4 + \bar{y}_1y_2\bar{y}_3\bar{y}_4 + y_1y_2\bar{y}_3y_4 + \bar{y}_1y_2y_3\bar{y}_4$$

This function L may be more efficiently realized using the method to be derived in sections 3.7-8.

3.3 SS Method:

The problem of determining minimum delay FSR realizations without

output logic for one, many, or all de Bruijn sequences of any degree n, is one of algebraically generating the de Bruijn sequence, then of realizing the hardware which will generate that particular sequence. Having a deterministic method of achieving both these steps will yield an algorithm suitable for a digital computer.

The single sequence (SS) method is included in the case studied in section 3.4, but deserves special attention as it is the most straightforward approach to generating a de Bruijn sequence. This method does not require a study of the sequences of degree n-1; it is a simple means of direct generation of a de Bruijn sequence of any degree n. This is accomplished by generating the sequence of period 2^n considering the words of length n, i.e. considering the recursive property.

The method considers words of length n-1, which by definition must occur twice in one period. Then starting with n 0's, follow each word of length n-1 by the 1 bit the first time, and the 0 bit the next time. Or conversely, start with n 1's and follow with 0's then 1's; this latter case yields the reciprocal sequence.

Example 3.3.1:

Consider the following sequences generated from n 0's:

for n = 3 : 00011101;

for n = 4 : 0000111101100101

Or, more explicitly, using a column notation whereby each n-bit word can be examined separately:

for n = 3 : 000

001

011

111

110

101

010

100

It is shown that the 1 bit is added until the word 110 in which the 0 bit had been added. This is so as the 111 word had already occurred.

Example 3.3.2:

Consider the following sequence generated from n 1's:

for n = 3 : 11100010

3.4 Overlap Method:

This method[10] of generating some de Bruijn sequences of degree n, is based on an arbitrary initial word and a function to be defined on all possible (n-1) bit words.

The method considers the sequence as being made up of a set of (n-1) bit words. Due to the recursive property, each (n-1) bit word will appear twice, once followed by the 0 bit and once followed by the 1 bit. The method is implemented using a "preference function"[12] (P), which indicates which bit follows the (n-1) bit word the second time; and, since it deals with binary sequences, the sequence is completely determined. This function P is determined by the r-overlap of each (n-1) bit word with the initial word, where r-overlap is defined as the maximum number of bits (r) of the end of the (n-1) bit word, which overlaps with the beginning of the initial word.

Example 3.4.1:

Consider the following r-overlaps:

(n-1) bit word	Initial word	r-overlap
010	0000	1
110	1111	0
101	1010	3
101	0100	2

As the r-overlap is defined from the (n-1) bit word onto the initial word, only the first (n-1) bits of the initial word are used. As there are only (2^{n-1}) different possible (n-1) length initial words, there are only as many possible preference functions. As each of these functions does not necessarily yield distinct sequences, the number of possible de Bruijn sequences, k, that can be generated is:

$$k \leq 2^{n-1} \qquad (3.4.1)$$

The algorithm develops as follows:

1. Choose an arbitrary initial word $(I_1, I_2, \ldots I_n)$.
2. Find the r-overlap (r) for all possible (n-1) bit words. Suppose the (n-1) bit word $(x_1, x_2, \ldots x_{n-1})$ has an r-overlap with $(I_1, I_2, \ldots, I_n)$; then the last r symbols of the (n-1) bit word, (i.e. $x_{n-r}, x_{n-r+1}, \ldots, x_{n-1}$) are the symbols $(I_1, I_2, \ldots, I_r)$.
3. Define the function P on the set of (n-1) bit words as follows:

$$P(x_1, x_2, \ldots, x_{n-1}) = I_{r+1} \qquad (3.4.2)$$

thereby gaining a preference function for all possible (n-1) bit words. But since the sequence Z will be recursive, each (n-1) bit word will, by definition, occur twice. Thus the

preference function will be applied as $\overline{P}$ the first time, i.e. the bit to be added to the sequence will be the binary complement of the bit defined by P. The subsequent time the (n-1) bit word appears the P function is used directly.

4. Generate the de Bruijn sequence $(z_1, z_2, \cdots, z_{2^n})$, as follows: the initial word $(I_1, I_2, \cdots, I_n)$ defines $(z_1, z_2, \cdots, z_n)$. Each new bit z_p, to be added to the sequence is defined by the preference function operating on the last (n-1) bit of the sequence generated at that moment:

$$z_p = (P)' \ (z_{p-n+1}, z_{p-n+2}, \cdots, z_{p-1}) \qquad (3.4.3)$$

where $(P)' = \overline{P}$ the first time and

$(P)' = P$ the second time the (n-1) bit word $(z_{p-n+1}, \cdots, z_{p-1})$ appears.

Example 3.4.2:

Consider generating the de Bruijn sequence of degree 4, starting with the initial word:

$$1010 = I_1 I_2 I_3 I_4.$$

Initially set up the (n-1) = 3 bit table to generate the r-overlap and the preference function:

(n-1) bit word $x_1\ x_2\ x_3$	r-overlap r	Preference function $P = I_{r+1}$
0 0 0	0	1
0 0 1	1	0
0 1 0	2	1
0 1 1	1	0
1 0 0	0	1
1 0 1	3	0
1 1 0	2	1
1 1 1	1	0

Now generate the de Bruijn sequence Z, considering the (n-1) bit word is made more obvious in tabular form:

	x_1 x_2 x_3	x_4
$z_1z_2z_3$ =	1 0 1	0 = z_4
	0 1 0	0 = z_5
	1 0 0	0 = z_6
	0 0 0	0 = z_7
	0 0 0	1 = z_8
	0 0 1	1 = z_9
	0 1 1	1 = z_{10}
	1 1 1	1 = z_{11}
	1 1 1	0 = z_{12}
	1 1 0	0 = z_{13}
	1 0 0	1 = z_{14}
	0 0 1	0 = z_{15}
	0 1 0	1 = z_{16}
	1 0 1	1 = z_1
	0 1 1	0 = z_2
	1 1 0	1 = z_3
	1 0 1	0 = z_4

The tabular form is a means of making the last (n-1) bits more obvious but is not required. The 16-bit sequence is more simply:

1010000111100101

3.5 Complete Method:

This method[12] will generate all the de Bruijn sequences. Using a similar method to the r-overlap case, the sequence is defined from an initial word and a preference function (P*) defined on the (n-1) bit

word.

This method is more complex as this function P* is defined from a preference function P_{n-2}, used in the generation of a de Bruijn sequence of degree (n-1), S_{n-1}, and from its corresponding feedback function F_{n-1}. Thus to develop an algorithm, it will be necessary to build sequences through every n, in order to have available the preference function and the feedback function of the previous case, as indicated in figure 3.5.1.

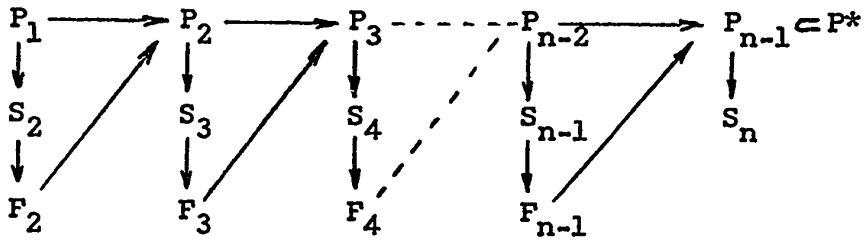


Fig. 3.5.1 Preference Function Buildup.

where P_i is the preference function defined on words of length i.

S_i is the sequence defined by P_{i-1} and is of length 2^i.

F_i is the feedback function which generates S_i.

The algorithm is as follows:

1. Choose an arbitrary initial word $(I_1, \ldots, I_n)$.
2. Let $P(x_1, \ldots, x_{n-2})$ be a preference function for the n-1 case, with the condition that:

   $$P(I_1, \ldots, I_{n-2}) = 1 \oplus I_{n-1} \mod 2; \qquad (3.5.1)$$

   and let F be its corresponding feedback function.

   (This preference function indicates which bit follows the (n-1) bit word the first time, which differs from the case in 3.4.)
3. Define function P* on the set of (n-1) bit words following the condition:

$$\left[P^*(x_1,\ldots, x_{n-1}) \oplus 1 \oplus F\right] \times \left[P(x_2,\ldots, x_{n-1}) \oplus F\right] = 0$$

(where the x is the cross-product) (3.5.2)

This condition will be met for a number of different (P_i^*) functions, each one defining a de Bruijn sequence.

As the initial word is defined, the following condition is determined:

$$P^*(I_1, I_2, \ldots, I_{n-1}) = 1 \oplus I_n \quad (3.5.3)$$

4. P*, as developed from equation 3.5.2, may not be fully determined for all (n-1) bit words; it may contain some "don't cares". Each possible combination of "don't cares" produces an individual de Bruijn sequence, hence, if there are d "don't cares" for a particular P_{n-1} and F_{n-1}, the number of sequences k the corresponding P* will produce will be:

$$k = 2^d \quad (3.5.4)$$

Therefore, individually, the sequences will be generated from the initial word and from $P_1^*, P_2^*, \ldots, P_k^*$. The other de Bruijn sequences can be generated from the other set of P* determined from the other P and F's.

5. Generate the de Bruijn sequence $(z_1, z_2, \ldots, z_{2^n})$, as follows: the initial word $(I_1, I_2, \ldots, I_n)$ defines $(z_1, z_2, \ldots, z_n)$. Each new bit z_p, to be added to the sequence, is defined by the particular preference function P_i^* (i = 1, 2, ..., k for each P, F case) operating on the last (n-1) bits of the sequence generated at the moment:

$$z_p = (P_i^*)' \ (z_{p-n+1}, z_{p-n+2}, \cdots, z_{p-1}) \qquad (3.5.5)$$

where $(P_i^*)' = P_i^*$ the first time and

$(P_i^*)' = 1 \oplus P_i^*$ the second time.

the (n-1) bit word

$(z_{p-n+1}, \cdots, z_{p-1})$ appears.

Example 3.5.1:

Consider generating the de Bruijn sequence of degree 4, starting with the initial word 1111 $(I_1 I_2 I_3 I_4)$ and having the preference function P for the (n-1) case, defined on the (n-2) length words, as follows:

(n-2) bit word x_1 x_2	P
0 0	0
0 1	0
1 0	1
1 1	0

This preference function meets the condition of equation 3.5.1 as

$$P(I_1 I_2) = 1 \oplus I_3$$

$$\therefore P(1\ 1) = 1 \oplus 1 = 0$$

The corresponding feedback function F, for the sequence defined by the above P and the initial word $(I_1, I_2, \cdots, I_{n-1})$ is:

$$F = 1 \oplus x_1 \oplus x_2 \oplus x_2 x_3$$

This function F is available to this algorithm. However, it will be shown that it can be derived from the P function (see fig. 3.5.1), using the methods to be described in sections 3.5 through 3.7.

The P* function is found by working with equation 3.5.2 on the (n-1) bit word, feedback function F and preference function P as shown in the following table:

x_1	x_2	x_3	F	P	$(F \oplus P)$	$(1 \oplus F)$	$(P^* \oplus 1 \oplus F)$	P^*	P_1^*	P_2^*
0	0	0	1	0	1	0	0	0	0	0
0	0	1	1	0	1	0	0	0	0	0
0	1	0	0	1	1	1	0	1	1	1
0	1	1	1	0	1	0	0	0	0	0
1	0	0	0	0	0	1	-	-	0	1
1	0	1	0	0	0	1	-	-	0	0
1	1	0	1	1	0	0	-	-	0	0
1	1	1	0	0	0	1	-	0	0	0

The above table demonstrates that the particular P and F functions used, yield P* with three dashes (don't cares) which allow for eight (2^3) cases, hence generating eight de Bruijn sequences, (after considering equation 3.5.3).

The sequence generated by

P_1^* is : 1111000010100110 and by

P_2^* is : 1111001010000110, the other sequences described by P_3^* to P_8^* can be similarly generated.

3.6 FSR Realization:

To realize the FSR which generates a desired de Bruijn sequence is to formulate its feedback function. This can be accomplished directly by building the FSR's assigned state table from the output sequence required; this table will then be the feedback function's table of combinations.

As the FSR required to generate the maximal sequence is cyclic with all states occurring in a single cycle, the state table, for a given degree n (i.e. with period 2^n), is constant for every case.

Example 3.6.1:

Consider the state behaviour of an FSR of degree 2, i.e. with $2^2 = 4$ states:

Present State	Next State
Y_1	Y_2
Y_2	Y_3
Y_3	Y_4
Y_4	Y_1

Fig. 3.6.1 State Table (degree 2)

Because the FSR's output z has been chosen to coincide with the feedback function (see fig. 2.1.4), and because this is also the next state bit for the first stage (see equation 2.1.3), this makes immediately accessible one of the state table's assignment columns, y_1'. Thus, this next state column is the exact replica of the FSR output, or of the de Bruijn sequence required.

The inherent bit shifting of the FSR allows the full assignment of the state table to be directly available, as each subsequent column is the previous column shifted through one bit.

Example 3.6.2:

Consider constructing the state table for the n = 3 case using the de Bruijn sequence: 11101000.

Present State			Next State			Present Output
y_3	y_2	y_1	y_3'	y_2'	y_1'	z
0	0	0	0	0	1	1
0	0	1	0	1	1	1
0	1	1	1	1	1	1
1	1	1	1	1	0	0
1	1	0	1	0	1	1
1	0	1	0	1	0	0
0	1	0	1	0	0	0
1	0	0	0	0	0	0

Fig. 3.6.2 Assigned State Table

The steps required to build this assigned state table are as follows:

1. The de Bruijn sequence is displayed in the output column, z, and also in the last next-state column, y_1'.
2. Each subsequent next-state column is similar to the previous one except that it has been delayed one unit of time. This defines all other next-state columns, y_i' for $i > 1$.
3. The constant state table property (as in fig. 3.6.1) dictates the assignment for all the present state columns. In this way, considering the internal states (see fig. 2.1.5) of the FSR, each next internal state is the present internal state delayed through one unit of time.

   In this manner the assigned state table is completely defined, thereby realizing the FSR circuit. The feedback function, thus defined by the y_1' column, is available considering this state table to be its table of combinations.

In this form the feedback function can be represented by the general sum of primary products (as shown in equation 2.1.8) and is thus in the first canonical form (FCAN).

Example 3.6.3:

Consider the assigned state table of example 3.6.2, the feedback function F in FCAN is thus,

$$F = \bar{y}_1\bar{y}_2\bar{y}_3 + y_1\bar{y}_2\bar{y}_3 + y_1y_2\bar{y}_3 + \bar{y}_1y_2y_3$$

3.7 EXOR Realization:

A method of designing a de Bruijn sequence generator has been presented. This can be accomplished by combining one of the introduced methods of algebraically generating a de Bruijn sequence (sec. 3.3-5), with the FSR realization method (sec. 3.6). The hardware realization of the generator will then be the actual implementation of the feedback function to the FSR.

The FSR realization method uses the FCAN form of the feedback function which can be unwieldy unless it is reduced whenever possible. Otherwise this form necessitates each term to consist of all n variables, including some in their complemented forms, which would require a great number of gates. A more convenient method than attempting to minimize such circuits, is to use the exclusive-or (EXOR) form (see equation 2.1.9), which requires only the non-complemented variables; furthermore, this form requires few variables per term, and, in this work (due to equation 2.5.2), would never require all variables occurring at once.

Thus, the EXOR case is considered as a more efficient means of economical realization, and a means of simply converting FCAN to EXOR would be of great use.

The Translation Matrix is such a means of converting the FCAN to the EXOR or vice-versa. This matrix uses the standardized 2^n tuples of

the canonical forms as described in section 2.1.

The matrix is an analysis of the secondary variable (y_i's) bit's distribution in the two canonical forms used. In the matrix, each row represents one binary variable of the EXOR-tuple, and each column one binary variable of the FCAN-tuple, as shown in figure 3.7.1. Thus, a unit entry at the intersection indicates that every secondary variable of the row is present in that particular column.

	Secondary Variables								
AND-OR	y_1	0	1	0	1	0	1	0	1	
Feedback	y_2	0	0	1	1	0	0	1	1	
Function	y_3	0	0	0	0	1	1	1	1	
	1	1	1	1	1	1	1	1	1	b_o
EXCLUSIVE-	y_1	0	1	0	1	0	1	0	1	b_1
OR	y_2	0	0	1	1	0	0	1	1	b_2
Feedback	$y_1\ y_2$	0	0	0	1	0	0	0	1	b_3
Function	y_3	0	0	0	0	1	1	1	1	b_4
	$y_1\ y_3$	0	0	0	0	0	1	0	1	b_5
	$y_2\ y_3$	0	0	0	0	0	0	1	1	b_6
	$y_1\ y_2\ y_3$	0	0	0	0	0	0	0	1	b_7
		a_o	a_1	a_2	a_3	a_4	a_5	a_6	a_7	

Fig. 3.7.1 Translation Matrix for n = 3.

The construction of this matrix, C, can be completed simply by considering each row, as one element of the EXOR-tuple, and filling in the bits according to the presence of the EXOR secondary variable in the respective columns, which are interpreted as the individual element

of the FCAN-tuple. Entry at c_{ij} will represent whether each secondary variable of b_i is present (in its uncomplemented form) in a_j.

Example 3.7.1:

Consider the entries at c_{45}, c_{33} and c_{63}. These represent the intersections of b_4a_5, b_3a_3 and b_6a_3 respectively. For the case of n = 3, a matrix of order 8 as shown in figure 3.7.1, the entries are as follows:

a) c_{45} has a unit entry as b_4 controls the single secondary variable entry of y_3 which occurs in the secondary variable entry of a_5 which is $y_1\bar{y}_2y_3$. It is noted that a_5 also controls y_1, but in this matrix entry, c_{45}, it is irrelevant.

b) c_{33} has a unit entry as b_3 controls the double secondary variable entry of y_1y_2, which both occur in the secondary variable entry of a_3, which is $y_1y_2\bar{y}_3$.

c) c_{63} has a zero entry as b_6 controls the double binary secondary variable entry of y_2y_3, of which only the y_2 variable occurs in the secondary variable entry of a_3. (The y_3 variable is not occurring in the necessary non-complimented form, and the y_1 need not be considered.)

Considering the simple construction principle, the method of conversion usage is quite systematic. To convert from the EXOR to the FCAN, or vice-versa, the method is as follows:

Consider the a_j-tuple (FCAN-tuple) as the row matrix A, the b_i-tuple (EXOR-tuple) as the row matrix B, and the translation matrix is the square matrix C.

Then the matrix equation, considering the sum to be the exclusive-or sum, is:

$$A = B \cdot C \qquad (3.7.1)$$

$$\text{i.e.} \quad a_i = b_o c_{oi} \oplus b_1 c_{1i} \oplus \ldots \oplus b_{2^n-1} c_{2^n-1\,i} \qquad (3.7.2)$$

$$\text{for } i = 0, 1, \ldots, 2^n-1.$$

This converts the FCAN-tuple to the EXOR-tuple; the reverse conversion, the a_j-tuple to the b_i-tuple is similar as the matrix is its' own inverse.

A further simplification of the feedback function conversion can be made by using Golomb's[12] function decomposition as described in section 2.5 and equations 2.5.1-2. Golomb has shown that for a cyclic FSR, the feedback function f can be simplified to a g-function as follows:

$$f(y_1, y_2, \ldots, y_n) = g(y_1, y_2, \ldots, y_{n-1}) \oplus y_n \qquad (3.7.3)$$

This decomposition means that the Translation Matrix can be reduced by a factor of two as there is no need to consider the 2^{n-1} factors of y_n. This is accomplished by using the matrix to find the EXOR g-function which is then translated to the required f-function by equation 3.7.3, i.e. by introducing the 1 bit in position $(2^{n-1} + 1)$, which represents y_n.

Example 3.7.2:

Consider the FCAN-tuple for the de Bruijn FSR of degree n = 4: 1111000100001110. Using the Translation Matrix, one can obtain the equivalent feedback function as the EXOR-tuple 1000100110000000 which includes no bit in any position above $(2^{n-1} + 1)$ as predicted by equation 3.7.3.

Thus the g-function simplification can be adopted by using the translation matrix to convert only the first half of the FCAN-tuple, namely 11110001 to the EXOR form, yielding the g-function which can be converted to the required f-function.

Using the Translation Matrix on the 2^{n-1}-tuple 11110001 yields the tuple 10001001. Then adding the $(2^{n-1} + 1)$ bit gives the EXOR-tuple 1000100110000000 as required.

Thus, it is now possible to mechanically generate the EXOR realization of the FSR for all possible de Bruijn sequences using the method developed in sections 3.5-7 consecutively. In this manner the method is programmable on a digital computer as will be shown in part 4.

3.8 Translation Matrix and its Algorithm:

The Translation Matrix, previously defined and used for canonical form conversions (section 3.7) can be of further use because it contains the truth table of the FSR (see fig. 2.1.5), when it is coupled to the FCAN-tuple. With this method, it is thus feasible to generate in a mechanical manner, the output sequence of the FSR. The method used to generate the sequence is based on the following equations:

a) The FSR state equation, no. 2.1.3:

$$y_1' = f(y_1, y_2, \cdots, y_n) \qquad (2.1.3)$$

b) The FSR feedback function, equation 2.1.8:

$$y_1' = a_o(\overline{y_1 y_2 \cdots y_n}) + a_1(y_1 \overline{y_2 \cdots y_n}) + a_2(\overline{y}_1 y_2 \overline{y_3 \cdots y_n}) + \cdots + a_{(2^n-1)}(y_1 y_2 \cdots y_n) \qquad (2.1.8)$$

c) The FSR's present state vector, equation 2.1.14:

$$Y_i = (y_1, y_2, \cdots, y_n) \qquad (2.1.14)$$

d) The FSR's next state vector, equation 2.1.15:

$$Y_i' = (y_1', y_1, y_2, \cdots, y_{n-1}) \qquad (2.1.15)$$

Because the feedback function y_1' depends only on the state of the secondary state variables (equation 2.1.3) at one instant of time, at

that instant, only one product term of the feedback function equation 2.1.8, is considered, as all others are zero; the one to be considered is the one which reflects the present state of the FSR, considering the secondary variable to be one in its non-complemented form and 0 in its complemented form. The mapping method introduced in section 2.1, to order the feedback function equation (2.1.8) product terms to the binary variables (a_j), can now be more simply interpreted from the present state vector. The numbering system for the binary variable subscripts is the state vector considered as a binary number when ordered from right to left.

Example 3.8.1:

Consider the following present states and the corresponding non-zero feedback function equation (2.1.8) term. Notice that the numbering scheme dictates that the present state, interpreted as a binary number, is the binary variable's subscript.

Present State Interpreted			
As Vector $y_4\ y_3\ y_2\ y_1$	As a Binary Number	Non-zero Equation term	Binary Variable
0 0 0 0	0	$a_o\ (\overline{y_4y_3y_2y_1})$	a_o
0 1 0 1	5	$a_5\ (\bar{y}_4y_3\bar{y}_2y_1)$	a_5
0 1 1 1	7	$a_7\ (\bar{y}_4y_3y_2y_1)$	a_7

In this way, it is now possible to rewrite the next state vector equation (2.1.15) as follows:

$$Y'_j = (a_j, y_1, y_2, \cdots, y_{n-1}) \qquad (3.8.1)$$

where j is found from the binary interpretation of the present state vector, i.e. $(j)_{10} = (y_{n-1}\ y_{n-2} \cdots y_1)_2$

From this equation it is possible to generate each subsequent state in the cycle, thus to generate the FSR truth table, and thereby generate the output sequence.

It is also apparent that, with the Translation Matrix and only the EXOR-tuple, it is also possible to generate the FSR output sequence. Equation 3.8.1 indicates that the generation of the state cycle is distinctly the generation of the a_j; furthermore, this binary variable is directly available from the j^{th} Translation Matrix column when multiplied with the EXOR-tuple vector.

Equation (3.7.2) is manifest and can be stated as:

$$a_j = \sum_{i=0}^{2^n-1} b_i \, c_{ij} \qquad (3.8.2)$$

considering the summing to be exclusive-or.

Using the above simple equation it is explicit to program a method to generate the output sequence of an FSR from its EXOR-tuple.

To be successful in programming a digital computer to solve either the conversion routine of section 3.7, or the EXOR-tuple output sequence generation as explained above, requires the storage of the $(2^n)^{th}$ order Translation Matrix. Obviously the size of this matrix expands inordinately as n increases and hence would require too great a storage memory for the solution of an n-stage FSR. In order to avoid this computer storage problem, an algorithm was developed which yields immediately the binary bit in any position of the matrix. The information required from the Translation Matrix for either of its above-mentioned uses, can be reduced to finding the matrix bit at the intersection of the EXOR-variable row (b_i) with the FCAN-variable or state column (a_j).

The algorithm is developed due to certain characteristics of the Translation Matrix. Initially apparent is the definite recursive properties of the Matrix. The basic matrix, for n = 1, is shown in fig. 3.8.1.

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Fig. 3.8.1 The Basic Matrix.

This basic matrix is the re-occurring structure throughout matrices of all order (see fig. 3.7.1). The introduction of an extra stage in an FSR is the introduction of an extra secondary variable which has the effect of doubling the possible number of terms in the feedback function, and hence increases the Matrix by the order of two. The building of the Translation Matrix is accomplished using the Basic Matrix replacing each unit by the complete matrix used in the previous (n-1) case. Thus, the n = 2 case matrix is constructed from the Basic Matrix whereby each unit is replaced by the Basic Matrix and the zero by a similar order all-zero matrix. Projecting this principle, by induction, the $(2^n)^{th}$ order matrix is again the Basic Matrix in which each unit replaced by the complete $(2^{n-1})^{th}$ order matrix, and the zero by an $(2^{n-1})^{th}$ order matrix of zeros.

The properties that allow the generation of the matrix to be a simple extension of the lower order matrix are the inherent property due to the peculiar tuple-ordering method. The algorithm which yields the individual bits of the matrix is founded on this projection principle coupled to two other properties: the main diagonal contains all ones, and all entries below the main diagonal are zeros. Thus to find the bit at the intersection of the row (R) and the column (C) the rules are: if the row number is greater than the column number, their intersection must be zero; if they are equal, it is one; if the row is less than the column, then

map this position on to its comparative position on the subsequent (2^{n-1}) matrix and re-apply the rules; if upon continuous remapping the order of the matrix becomes one, then obviously the bit is one.

The rules are:

a) if $R > C \longrightarrow 0$

b) if $R = C \longrightarrow 1$

c) if $C > R \longrightarrow$ project to map of next smaller degree and re-apply rules.

When map degree is 1, then $C > R \longrightarrow 1$.

This algorithm lends itself conspicuously to binary representation as the matrix orders are of the power of two. Hence, mapping of the R and C representations onto one of degree lowered by a power of two is essentially an upper binary bit strip. This expresses the power of the algorithm when used with the binary digital computer.

Example 3.8.2:

Consider the following row and column designations, and the subsequent intersection bit:

1. $R = 7$

   $C = 7$ $\therefore$ $R = C \longrightarrow 1$ i.e. $c_{7,7} = 1$

2. $R = 8$

   $C = 5$ $\therefore$ $R > C \longrightarrow 0$ i.e. $c_{8,5} = 0$

Consider the following cases which require an upper bit strip after the binary interpretation:

3. $R = 10 = 1010_2 \longrightarrow 010 \longrightarrow 10$

   $C = 13 = 1101_2 \longrightarrow 101 \longrightarrow 01$

   now $R > C \longrightarrow 0$ i.e. $c_{10,13} = 0$

4. $R = 8 = 1000_2 \longrightarrow 000 \longrightarrow 00 \longrightarrow 0$

   $C = 15 = 1111_2 \longrightarrow 111 \longrightarrow 11 \longrightarrow 1$

now as map is of degree 1

and $C > R \longrightarrow 1$ i.e. $c_{8,15} = 1$

5. $R = 5 = 0101_2 \longrightarrow 101$

$C = 13 = 1101_2 \longrightarrow 101$

now $R = C \longrightarrow 1$ i.e. $c_{5,13} = 1$

It is now possible, with the algorithm described above, to program the conversion method developed in section 3.7 while requiring very little computer memory. It is with this procedure that the computer programs of part 4 were developed.

4. DIGITAL COMPUTER PROGRAM DISCUSSION

4.1 Introduction

It would be highly desirable to implement the algorithms of part 3 into useful digital computer programs. This has been done and these are block diagrammed and listed in the appendices. However, it was felt that certain descriptions of the programming approaches and details would probably facilitate any attempts of understanding or modifying the programs. It is in this view that the following discussion sections are presented.

The discussion initially deals with the general programming problem; then the individual programs are discussed in the subsequent sections.

The leading problem was the choice of the computer language to be used. Normally this choice is dictated by the nature of the problem to be solved.

The methods of presentation and of solution of the algorithms, are in the form of binary data and in its manipulation. To withdraw the most efficient program to solve binary data problems, one should

resort to the assembly level of language, as most high-level languages are designed to deal with decimal (base 10) numbers.

The dilemma is heightened as such assembly-level languages are compatible only to one model-type of computer or to one system-type of computer within a single manufacturer's realm. And, as it was the intention of the author to allow a more universal degree of compatability without restriction to a system or any one manufacturer, the program was adapted to the high-level language, Fortran, with certain compensating additions.

In order to process the individual bits of binary data in Fortran, a set of five simple assembly sub-routines was set up in order to allow the right or left shifting of binary bits within a memory word. In this way, the programs are fully compatible to any computer system with a Fortran compiler, with the exception of the sub-routines which can simply be written into the system to be used, as will be explained in section 4.5.

The writing of a computer program always entails a certain degree of compromise. The main concession is the choice between processing speed and memory requirements; in most complex programs there exists this inverted relationship, where one can increase the speed by using more memory or do the reverse. As the cost of computer operating time is high and as the available memory is finite and does limit the size of the problem to be solved, a compromise must take place.

The necessity of such a compromise is exemplified in the methods chosen which would allow access to the individual binary bits of the word. As the IBM 360/40 computer used consisted of 32 bits or 8 byte words, it was decided that using individual words or bytes for the storing of single bits was ludicrous; this thought was further

substantiated in the light of the requirement of 2^n bits and the desire to process problems of large n. The solution was to use computer words to store the complete binary word.

The next problem was access to binary bits. The idea of setting up a system of masks was discarded in favour of shifting routines. As the high-order bit of the 32 bit words is used as the sign-bit, herein lay a simple means of checking the upper bit by testing the arithmetic sign of the computer word. Thus one could check each bit of a word by subsequent left shifting it into the sign-bit position. The shifting method does have an alternative: either the hardware shift instruction (available only in assembly-level) can be used, or the arithmetic multiplication by two would have similar effect. However, the arithmetic instruction could adversely affect the word in certain systems: in the case of attempting to move a bit into the sign position, it could cause an overflow situation, or defeat the anticipated use of the upper-bit position, and, in certain cases, give an erroneous bit reading by giving the proper sign. Such problems are avoided by using the assembly-level shift routines when upper-bit shifts are required, and using the arithmetic multiplicative shift for low-order (subscript generation) situations. (Both systems are used concurrently in the "generate new word" section of the "Complete Method" program.)

Thus, binary words are stored in computer words for purposes of manipulation and storage efficiency. A different system is used for output purposes, as binary outputs would prove unwieldy and lengthy. The output required on demand can consist of the de Bruijn sequence, the FCAN or the EXOR realization function or any combination of these three. Different output methods are available for either the sequence output or the function output.

In the case of the sequence output, a direct hexadecimal output is available as used in the program A; the alternative method, used in the other two programs, consists of grouping the similar bit strings and using the digit representing the length of each such string. Thus, the sequence will be represented by a string of digits.

Example 4.1:

Binary Sequence : 0111 0010 1100 0101

is in hexadecimal form : 72C5

is in digit string form : 1 3 2 1 1 2 3 1 1 1

The realized functions can be represented as a truth table vector which would be more compact, but the method used in the programs is more readily interpreted and therefore was judged more desirable. In these programs the function forms were presented as a set of hexadecimal words each representing binarily one of the products of the internal states. Thus, the FCAN function is the inclusive-or summation of the hexadecimal word products while the EXOR function is the exclusive-or summation. These outputs are so titled. The hexadecimal words differ slightly in each case as they are right-adjusted in the EXOR case and left-adjusted in the FCAN case while neglecting the high-order bit. This is shown in the following general forms: the FCAN function binarily represents $(0, y_n, y_{n-1}, \ldots y_1, 0, \ldots 0)$, while the EXOR function represents $(0, \ldots , 0, y_n, y_{n-1}, \ldots y_1)$.

Example 4.2:

In the case of n = 5:

FCAN product : 5C000000 and

EXOR product : 0000000D would represent

binarily 0101, 1100, 0, ..., 0 and 0, ..., 0, 1101

respectively, which is interpreted as

$y_5\bar{y}_4y_3y_2y_1$ and $y_4y_3y_1$.

For ease of interpretation the outputs of the last two programs are titled with this general form.

The methods of programming each of the three algorithms will be explained in the following sections.

4.2 Programming the S.S. Method: (Program A)

The program using the SS method was approached in three sections which became the guide for the subsequent two programs. The sections are: the internal generation of the de Bruijn sequence, the FCAN FSR realization, and the translation to the EXOR FSR realization.

The first section is simplified by dividing the de Bruijn sequence into individual left adjusted n- bit words which are stored in the vector MT. The program begins with the all 0-bit word and produces the next word by shifting out the high-order bit and introducing a new low-order bit. The method is to temporarily assume the new bit to be 1 then to scan the MT-vector for this temporary word. If such a word is present, the new bit is changed to a zero and the word is inserted in the MT-vector. Otherwise the word is directly included and the procedure is repeated till the all 0-bit word is regenerated. At this point the algorithm is completed and the de Bruijn sequence is available in the output (MO) vector.

The MT-vector is now interpreted as the present state vector of the assigned state table. The next-state vector is directly available as the MT-vector shifted through one position, due to the inherent cyclic characteristic of the FSR. Furthermore, because the feedback function (y_1', also the present output z) is present in the next state vector, it is available through shifting; hence, a different MT-vector is built up by selecting the words which precede (i.e. which are from the present state) each feedback function bit in the next state vector.

This new vector constitutes the FCAN function table.

The third section is an implementation of the Translation Matrix Algorithm using the maximal FSR's g-function. The MT-vector is now used to simplify the FCAN function into the g-function by discarding all terms which are a function of y_n, represented by a 1 in the high-order bit position. This g-function table is then used by the algorithm to generate the MX-vector. This vector is the EXOR-function vector which will then, in turn, be used to generate the EXOR-function terms, available in the new MT table.

4.3 Programming the Overlap Method:

Programming the r-overlap method is accomplished in 3 main new steps.

The first step consists of choosing the arbitrary initial word (IWD). This is accomplished for each of the NA sequences requested. The program starts at the IWD, consisting of all zeros and sequentially incrementing it as more are required. Each of the IWD bits is made available by its individual storage into the 'MR-vector' table, made possible by left shifting (SHFT1) IWD into the upper left or sign bit. Thus the r^{th} bit is available as the $(r + 1)^{th}$ term of the vector; the vector is shifted 1 unit as Fortran storage procedure disallows a 0^{th} term in a vector.

As the algorithm will define the second preference function, this method is mutated by actually storing the binary inverses of the IWD bits into the 'MR-vector'.

The next step is the definition of the "preference function" (P) in the 'MX table' by the r-overlap method. This is accomplished by choosing one of the ($NMT = 2^{n-1}$) (n-1) bit words and comparing it to the IWD in a series of 'strip compare' steps. After each unsuccessful

compare, IWD is put through a right bit strip (SHFT4) and the (n-1) bit word through a left bit strip (SHFT2), both strips leaving the words left-adjusted, till they are equal (i.e. compare successfully), then the equivalent r^{th} bit is stored in the 'MX table', subscripted in accordance with the (n-1) bit word interpreted as a number. This process is repeated for each of the (n-1) bit words, till the 'MX table' is complete.

The final step is the generation of the new word (NWD) from the last word (LWD). This is accomplished by left shifting and upper bit stripping of the LWD (in the exact parallel manner that the shift register will operate), then right shifting the remaining (n-1) bit word to the low order, to be used as the adjusted subscript to locate the new bit from the 'MX table'. At this time, after insertion of the new bit (thus forming NWD) this new bit is inverted and stored, in order to have available the second P-function.

The FCAN functions are also defined in this loop and stored in the 'MT table', as the presence of a low order unit (actually the feedback function) in NWD indicates that LWD is one of the functions.

4.4 Programming the Complete Method:

The "complete method" program consists of a preference function approach, as in the previous method.

This method's preference function is defined from the previous (n-1) level function and feedback function, therefore another algorithm must now be incorporated, in order to define the required preference function.

Once this function is available, the program follows similar steps to the circuit realization as did the previous case.

The other addition is due to a "register system" required to allow

the generation of all the de Bruijn sequence circuits.

Diagram 4.1 indicates the preference function algorithm showing the progressive chain-type build-up using the previous preference function P_i of degree i and the necessary feedback function F_i defined from the sequence S_i.

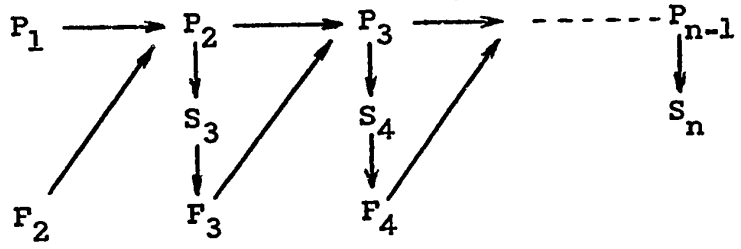


Fig. 4.1 Preference Function Algorithm

Thus, from a defined P_1 and F_2, we can derive P_2 which allows the generation of S_3 from which we can extract F_3. Then the process is algorithmically repeated till the desired P_{n-1} is derived.

As each preference function is a binary table and includes a set of "don't cares", each preference function is in actuality a set of preference functions, each generating a different sequence. This problem is dealt with using a "Register System" in order to avoid what would be a prohibitive storage problem if one were to attempt to store the preference functions in the computer.

The "Register System" is a set of vectors of the "don't cares" from the preference function tables. This system requires only one such vector (or register set) for each level required (e.g. for generation of level n = 10, only 10 vectors are needed). The registers contain only the "don't cares", and hence allow much more compact storage , greatly simplifying the method of keeping track of which "don't cares" cases were used. This latter problem is solved using the "register increment system". This increment system dictates

that the "don't care" vector begins in its all-zero case, then increments the vector through all possible cases to the all-one case.

The method defined to keep track of the different lengths of these "don't care" vectors is called the set of increment registers. Each of these registers consists of a properly placed binary bit such that the left-adjusted "don't care" corresponding register is of the exact vector length as determined by the number of "don't cares" in that particular case.

The computer program uses the preference function algorithm as follows:

1. The next level preference function is built up in vector MXT using (-1) to store "don't cares", MXT is derived from the previous preference function stored in vector MX and the feedback function in vector MF. To overcome the problem of vector MX being only half the length of vector MF, a marker IMP allows for the proper double use of the MX variables.
2. Vector MX is then defined from vector MXT using the corresponding preference function register MR. After usage of the MR, it is incremented using register KRN which had been defined from the "don't care" counter KSE.
3. The marker MMK indicates when the preference function has reached the (n-1) level, P_{n-1}, and that the preference function algorithm is completed. This will freeze F_{n-1} in order to be able to generate all the S_n's possible at this point without having to regenerate all the preference functions for each sequence.

4. After all sequences possible have been generated for the P_{n-1} defined, a new one is defined. This is accomplished by re-deriving it from the preference function algorithm using the incremented registers, thus producing a different set of sequences.

   This method is continued, generating all sequences for each set of P_{n-1}, and deriving all possible P_{n-1}'s using the register increment system. When all possible combinations of P_{n-1} have been used, all possible sequences have been found.

4.5 Required Sub-routines:

As explained earlier, as a means of using memory space efficiently, whole binary words are stored in computer words. Thus, in order to have access to the individual binary bits, shifting sub-routines were incorporated to the Fortran program. Shifting is used for two bit manipulations: it is used to single out one bit in the high-order or sign position in order to test it and thereby identify it; it is used to remove a definite number of bits by shifting out the controlled number of bits off either end.

The sub-routines used are such that the thirty second or upper bit is the sign bit and that bits are lost if shifted off either end, i.e. through right or left shifting. If the Fortran program is to be used with any other computer system, sub-routines must be added and be written in such a way as to resolve the same effects; for example, the CDC 3200 system allows bits to be shifted off one end only, while shifting off the other end results in a wrap-around effect, thereby not losing the bits.

The sub-routines written and used in this IBM 360/40 application

are basically a variable left shift SHFTL and a variable right shift SHFTR with the variable in the brackets following the word to be shifted.

Example 4.3:

CALL SHFTL (MSR, N) will cause the word MSR to be left shifted through N bits.

Essentially these are the only sub-routines required but, in order to save processing time, other sub-routines were added, which would accomplish a certain combination of shifts required at that particular time, but which could be accomplished by the proper combination of SHFTR and SHFTL.

These specialized shifts originate due to the left-adjusted words which are such as to keep the upper or sign bit empty. Thus, SHFT2 is a left bit strip which returns the new word to the proper left adjustment; this requires a left shift through two bits, followed by a right shift through one bit. SHFT4 is its complement. It is a right bit strip which returns the new word to the proper left adjustment but, in this case, a variable is needed to determine how far to the right the word should be shifted before it is to be brought back. This instruction allows the variable right shift followed by a left shift through the same variable. The other sub-routine, SHFT1, is simply a left shift through one bit requiring no variable.

Example 4.4:

Consider the following binary words defined in the upper 8 bit positions, positions 25 through 32:

Computer Word	Bit Position: 32	31	30	29	28	27	26	25	...	1
A :	0	1	0	1	1	0	0	0	...	0
B :	0	0	1	1	0	0	0	0	...	0
C :	0	1	0	1	0	0	0	0	...	0
D :	0	0	0	1	0	1	1	0	...	0
E :	0	1	0	1	1	0	0	0	...	0
F :	1	0	1	1	0	0	0	0	...	0

B would be the result of CALL SHFT2 (A)

C would be the result of CALL SHFT4 (A, K)

if K = 29

D would be the result of CALL SHFTR (A, L)

if L = 2

E would be the result of CALL SHFTL (D, L)

F would be the result of CALL SHFT1 (A)

5. CONCLUSION

Digital computer programs are presented which are now the tool that makes available, for any n, the set of FSR internal feedback logics which will generate, upon demand, one, some, or all of the de Bruijn or the non-linear maximal sequences. Furthermore, these logics are given, upon command, either as products of inclusive or exclusive or-gates.

This further choice is possible as a translation method was developed (in both matrix and algorithmic form) which directly transforms the inclusive-or canonical form to the exclusive-or canonical form, or applied backwards, will accomplish the reverse conversion; this method was then incorporated in all the programs.

A background section was included to define the fundamentals and outline the evolutionary developments of the theory and contains a brief survey of the linear theory.

Certain alternative methods which yield the required sequences are presented, including the scheme using output logic.

As the final result is the set of three digital computer programs, a section describing the approaches used and explaining their methods, is also contained in this work.

REFERENCES:

1. Birkoff, G. and MacLane, S. (1961), A Survey of Modern Algebra. The Macmillan Co., New York, N.Y.; 1961.

2. Bryant, P. R. and Killick, D. R. (1962), "Nonlinear Feedback Shift Registers". IRE Trans. Electron. Computers (Correspondence), vol. EC-11, pp. 410-412; June, 1962.

3. Brzozowski, J. A. (1965), "An Essay on Feedback". Department of Electrical Engineering, University of Ottawa, Ottawa, Ontario, Technical Report No. 65-2; March, 1965.

4. Brzozowski, J. A. and Davis, W. A. (1964), "On the Linearity of Autonomous Sequential Machines". IEEE Trans. Electron. Computers, vol. EC-13, pp. 673-679; December, 1964.

5. Davis, W. A. (1965), "On Shift Register Realizations for Sequential Machines". 1965 IEEE Conf. Record on Switching Cct. Theory and Log. Design, pp. 71-83; October, 1965.

6. Davis, W. A. (1966), "Generation on Delayed Replicas of m-sequences". Proc. IEE, vol. 113, pp. 295-6; February, 1966.

7. de Bruijn, N. G. (1946), "A Combinatorial Problem". Proc. Koniklyke Nederlandsche Akademie van Wetenschappen, Amsterdam, vol. 49, part 2, pp. 758-764; June, 1946.

8. Edwards, R. W. (1963), "Algebraic Synthesis of Switching Networks". Stanford Electronic Laboratories, Stanford University, Stanford, California, Technical Report no. 2205-1, SEL-63-029; April, 1963.

9. Elspas, B. (1959), "The Theory of Autonomous Linear Sequential Networks". IRE Trans. on Circuit Theory, vol. CT-6 no. 1, pp. 45-60; March, 1959.

10. Ford, L. R. (1957), "A Cyclic Arrangement of n-tuples". The Rand Corporation, Santa Monica, California, Report no. P-1070; April 23, 1957.

11. Golomb, S. W. (1955), "Sequences with Randomness Properties". The Glenn L. Martin Co. (now Martin-Marietta Corporation), Baltimore, Maryland, Final Report on Contract no. SC-54-333611; June, 1955.

12. Golomb, S. W. and Welch, L. R. (1957), "Nonlinear Shift Register Sequences". Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, Memorandum no. 20-149; October 25, 1957.

13. Golomb, S. W., Welch, L. R., and Goldstein, R. M. (1959), "Cycles from Nonlinear Shift Registers". Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, Progress Report no. 20-389, Ordcit Project Contract no. DA-04-495-ORD18; August 31, 1959.

14. Golomb, S. W. (1967), Shift Register Sequences*. Holden-Day, Inc., San Francisco, California; 1967.

15. Good, I. J. (1946), "Normal Recurring Decimals". The Journal of the London Mathematical Society, vol. XXI, part 3, no. 83, pp. 167-169; July, 1946.

16. Huffman, D. A. (1955), "The Synthesis of Linear Sequential Coding Networks". Proc. Third London Symposium on Information Theory, pp. 77-95; September, 1955.

17. Kautz, W. H. (1958), "State-logic Relations in Autonomous Sequential Networks". Proc. Eastern Joint Comp. Conf., Philadelphia, Pennsylvania, p. 119; December, 1958.

18. Kautz, W. H., Elspas, B., and Stone, H. (1961). "Advanced Computer Design Theory". Reports nos. 3131 and 3638, Stanford Research Institute, Menlo Park, California; January and November, 1961.

19. Kiyasu, Z. and Toda, I. (1963), "Maximal Period Feedback Shift Registers". Review of the Electrical Communication Laboratory, vol. 11, nos. 1-2, pp. 65-93; January-February, 1963.

20. Magleby, K. B. (1963), "The Synthesis of Nonlinear Feedback Shift Registers". Stanford Electronic Laboratories, Stanford University, Stanford, California, Technical Report no. 6207-1, SEL-63-118; October, 1963.

---

*(14) includes (11), (12) and (13).

21. Marsh, R. W. (1957), "Tables of Irreducible Polynomials over GF(2) through Degree 19". U.S. Department of Commerce, O.T.S.; October 24, 1957.

22. Peterson, W. W. (1961), Error-Correcting Codes. The M.I.T. Press, Cambridge, Massachusetts and John Wiley & Sons, Inc., New York, N.Y.; 1961.

23. Reed, I. S. (1963), "A General Viewpoint of Shift-Register Sequences". The Rand Corporation, Santa Monica, California, Report no. RM-3874-PR; October, 1963.

24. Reed, I. S. and Turn, R. (1966) "General Shift-Register Sequence Generators". The Rand Corporation, Santa Monica, California, Report no. P-3473; November, 1966.

25. Rees, D. (1946), "Note on a Paper by I. J. Good". The Journal of the London Mathematical Society, no. 83, vol. XXI, part 3; July, 1946.

26. Roberts, T. A. (1963), "Analysis and Synthesis of Linear and Non-linear Shift Register Generators". International Telemetering Conference, pp. 390-399; October, 1963.

27. Scholefield, P. H. R. (1960), "Shift Registers Generating Maximal Length Sequences". Electronic Technology, p. 389; October, 1960.

28. Srinivasan, C. V. (1962), "State Diagrams of Linear Sequential Machines". Journal Franklin Institute, vol. 273, pp. 383-418; May, 1962.

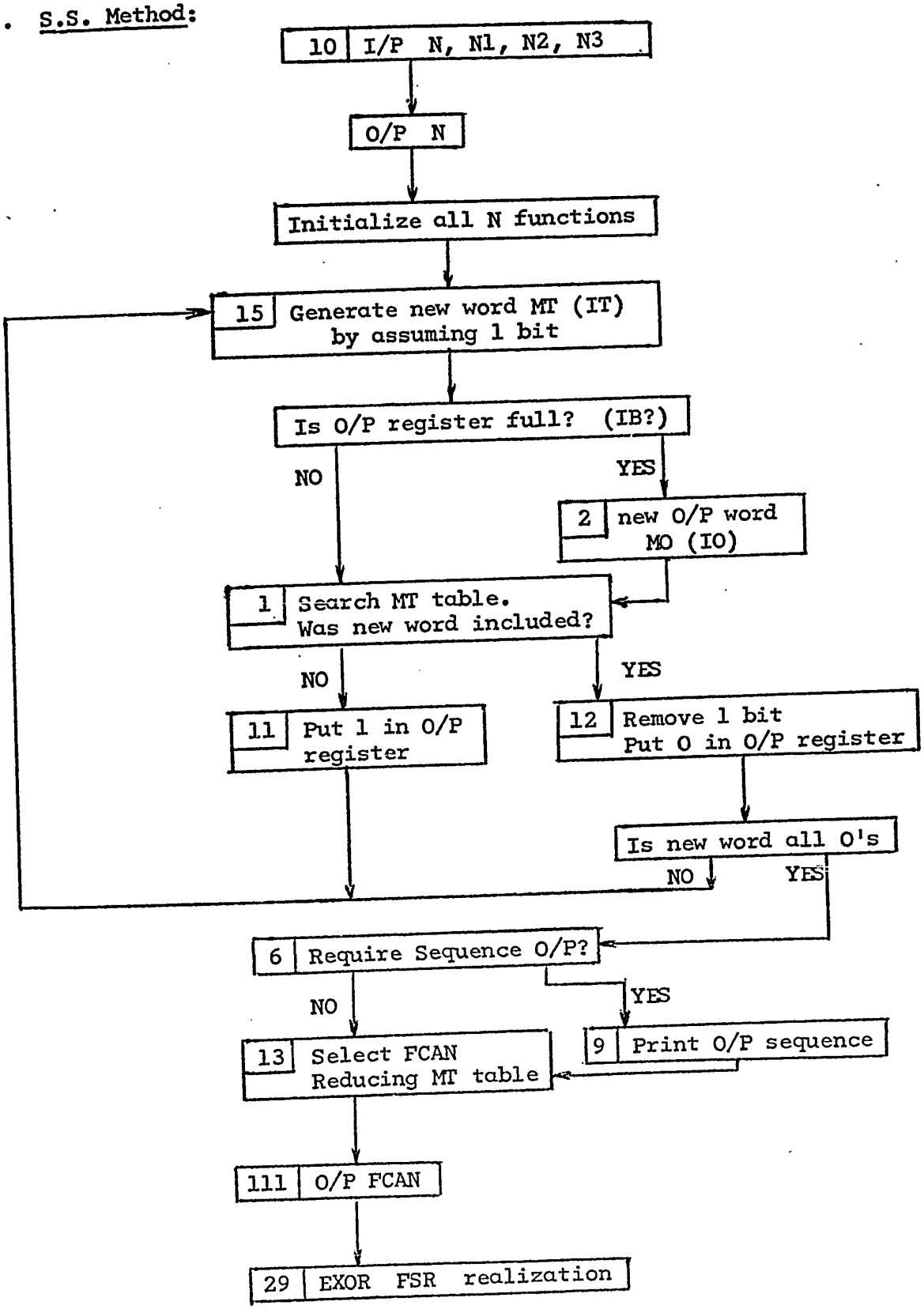
29. Young, F. H. (1958), "Analysis of Shift Register Counters". Journal A.C.M., vol. 5, pp. 385-388; October, 1958.

30. Zierler, N. (1955), "Several Binary Sequence Generators". M.I.T., Lincoln Laboratory, Lexington, Massachusetts, Technical Report no. 95; September 12, 1955.

31. Zierler, N. (1959), "Linear Recurring Sequences". Journal Soc. Indust. Appl. Math., vol. 7, no. 1, pp. 31-48; March, 1959.

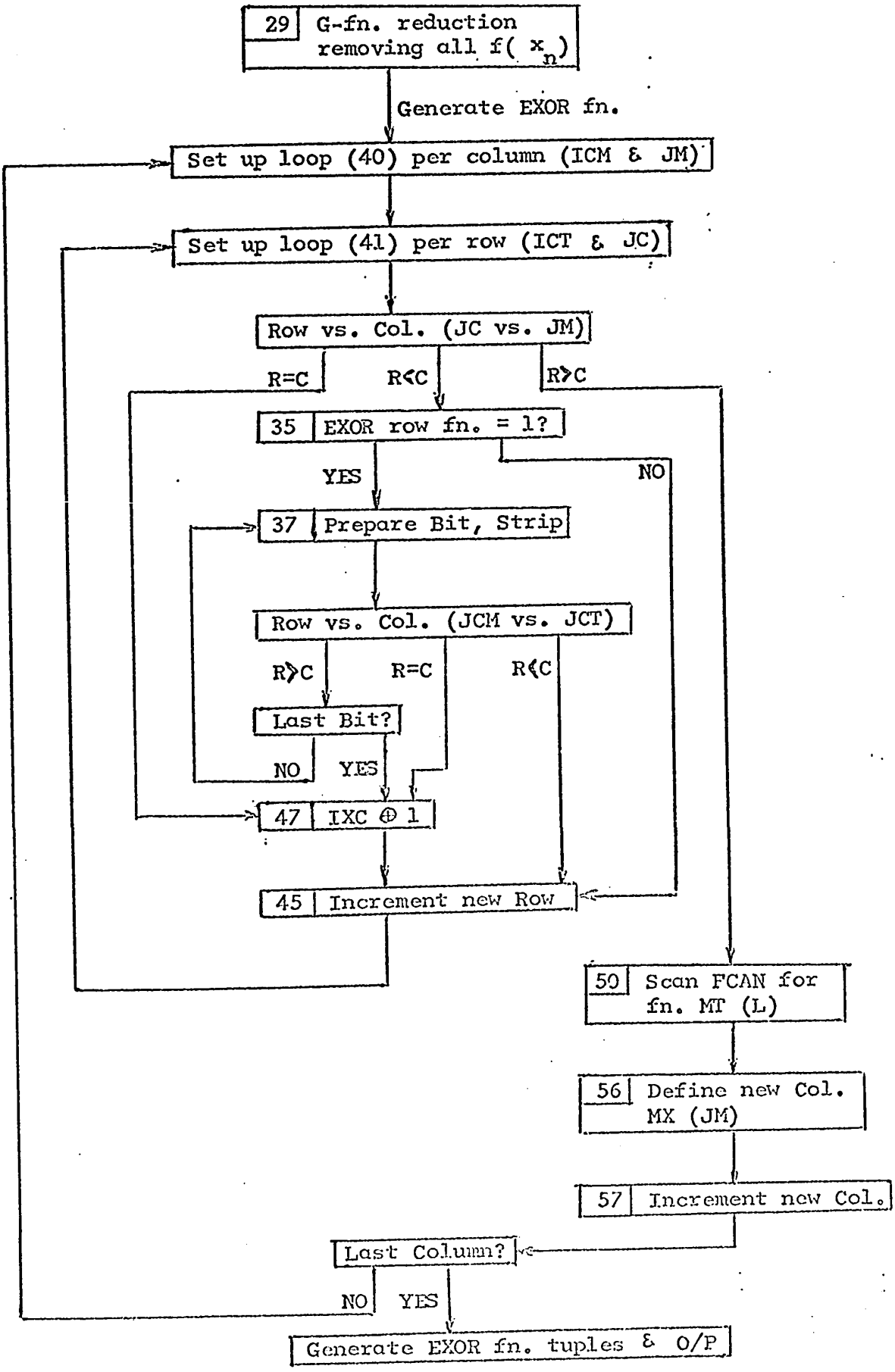
APPENDIX I: BLOCK DIAGRAMS

1. S.S. Method:

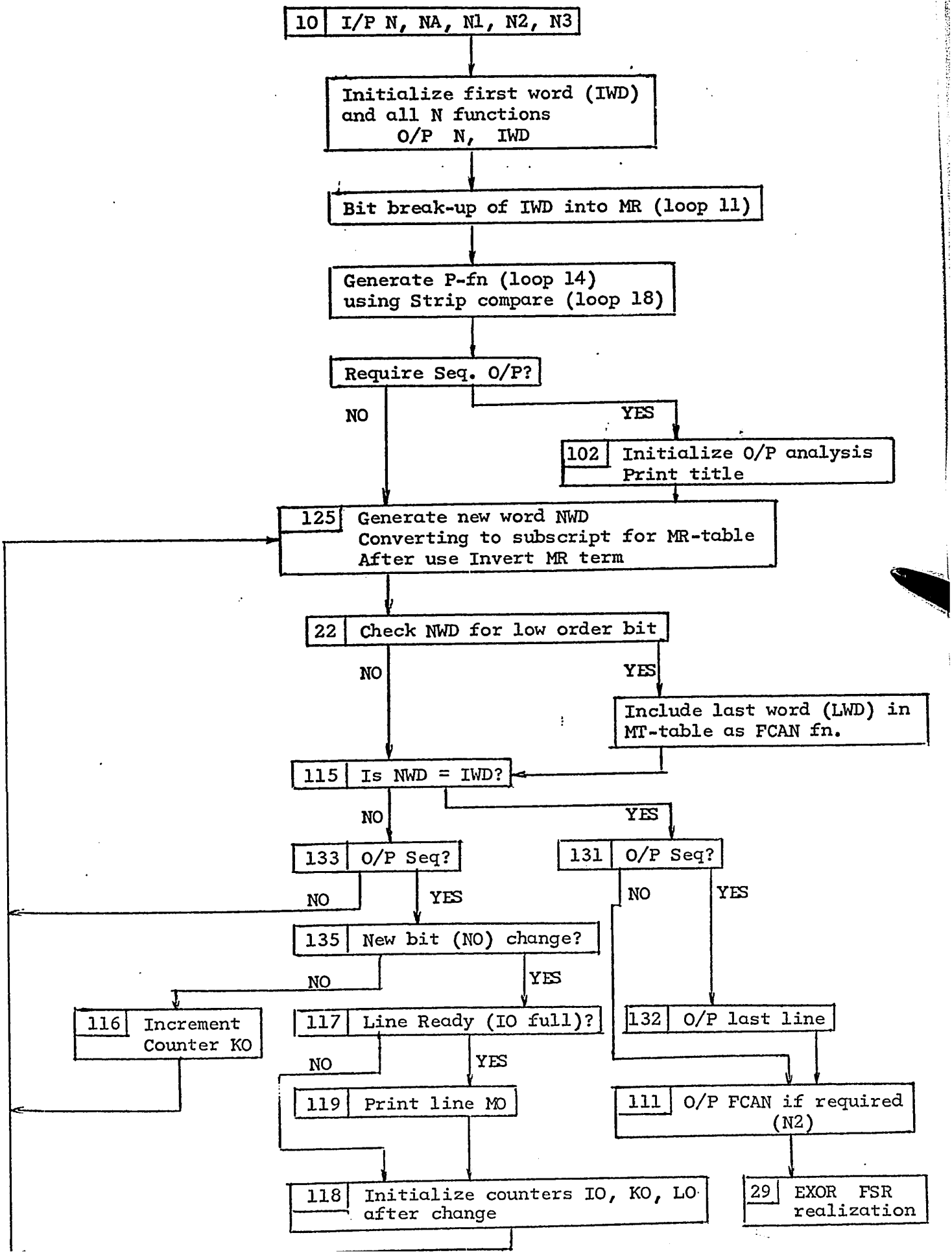


2. Translation Matrix Algorithm:

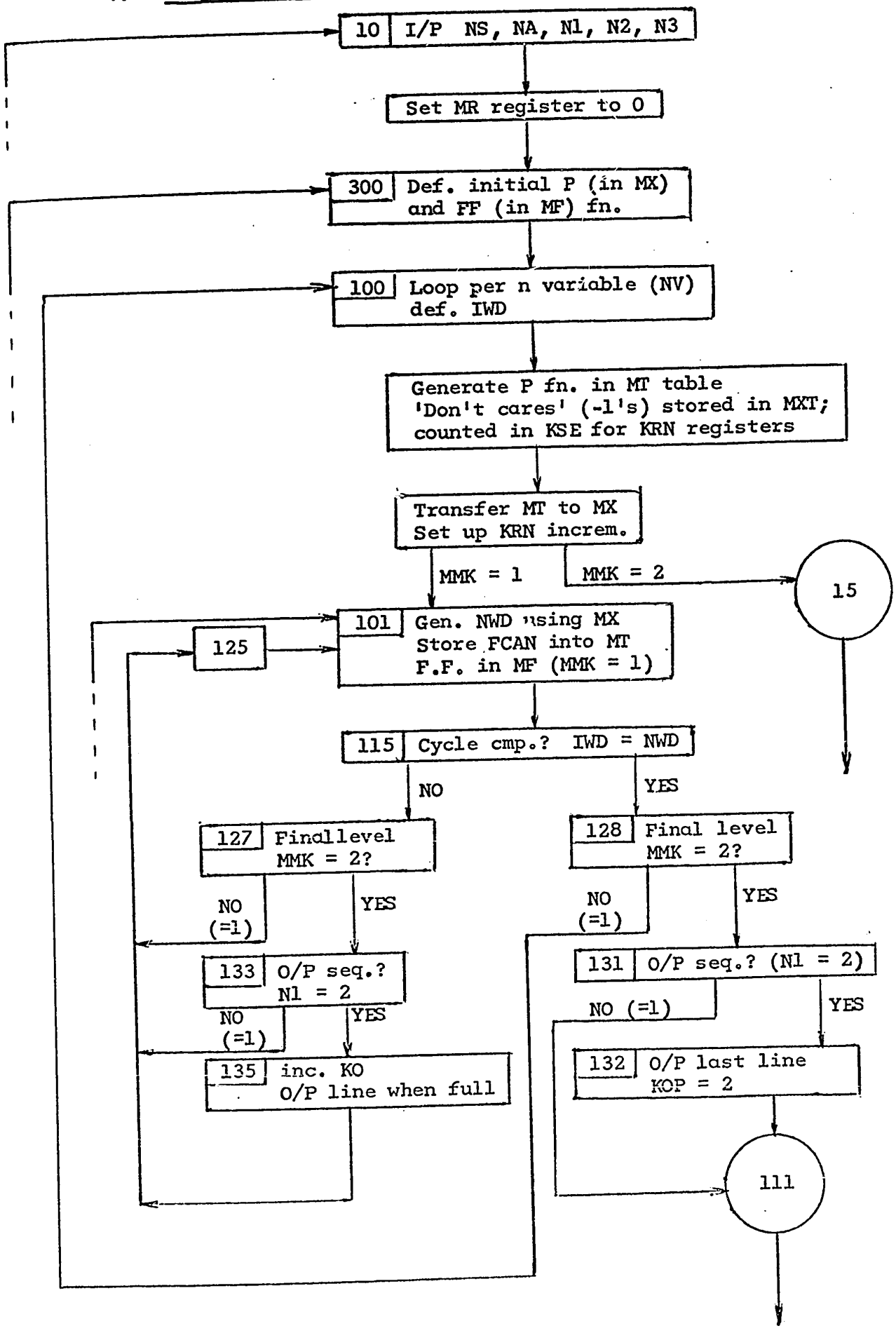
EXOR FSR realization

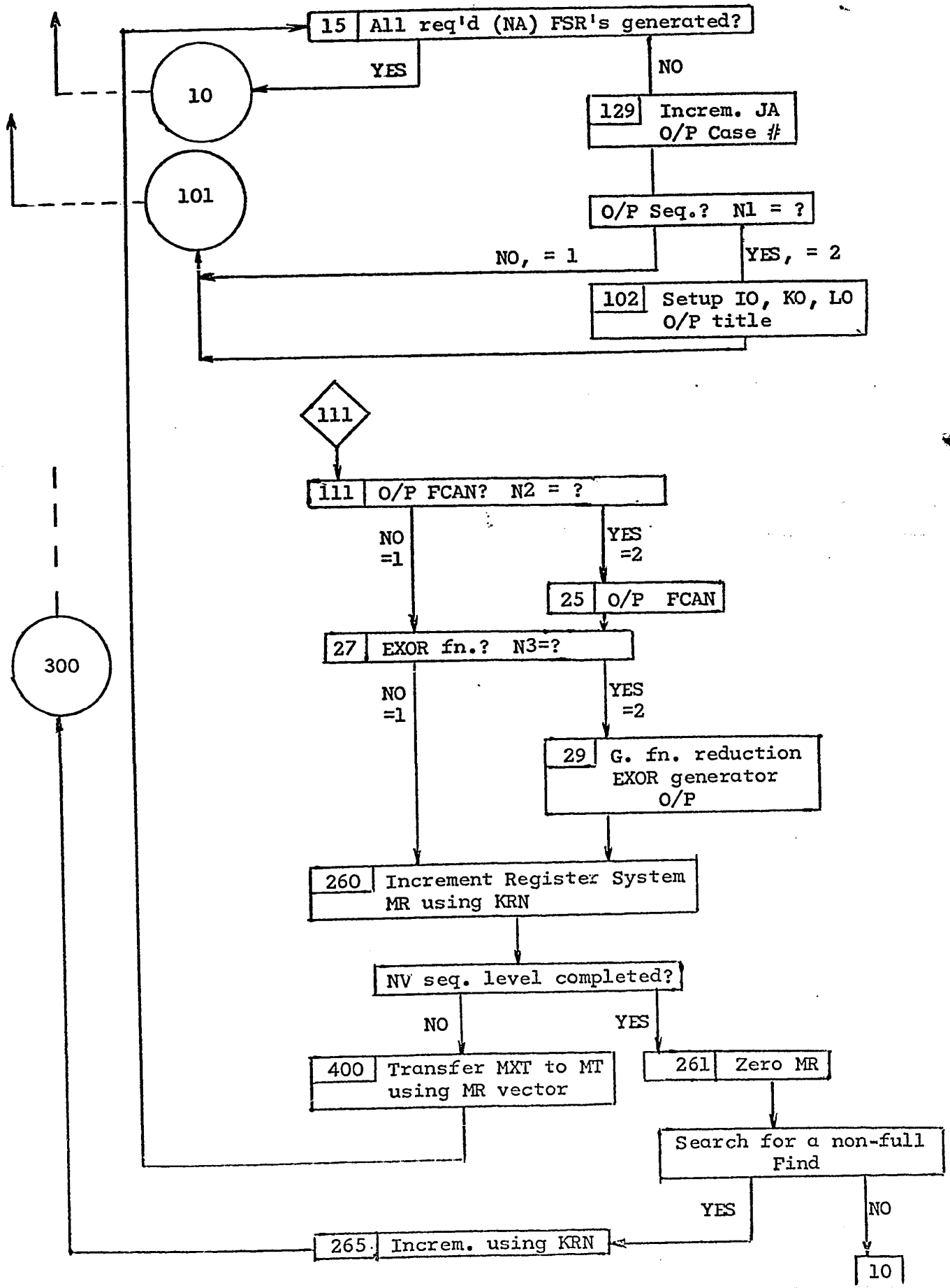


3. r-overlap Method:



4. Complete Method:




15 All req'd (NA) FSR's generated?
YES
NO
10
129 Increm. JA O/P Case #
101
O/P Seq.? N1 = ?
NO, = 1
YES, = 2
102 Setup IO, KO, LO O/P title
111
111 O/P FCAN? N2 = ?
NO =1
YES =2
25 O/P FCAN
27 EXOR fn.? N3=?
NO =1
YES =2
29 G. fn. reduction EXOR generator O/P
300
260 Increment Register System MR using KRN
NV seq. level completed?
NO
YES
400 Transfer MXT to MT using MR vector
261 Zero MR
Search for a non-full Find
YES
NO
265 Increm. using KRN
10

APPENDIX II: PROGRAMS' METHOD OF USE, INPUT CARDS

Input to the programs is accomplished using the card reader and the output uses the line printer. Each program requires a single input card which must yield the degree of the machine to be generated and three output indicators depending on what output is desired.

The Fortran Format requirements dictate that these input numbers must be stringently placed in the proper card columns (cc) as will be indicated.

Furthermore, all numbers must be right adjusted (as these are fixed-point numbers). The format for each input card is given as follows:

1. S.S. Method Program:

cc 1 and 2 : N, the degree of the de Bruijn sequence or of the machine required.

cc 3 : N1, the binary indicator (0 or 1) which dictates whether or not the user wishes to include the actual de Bruijn sequence in the program output.

cc 4 : N2, the binary indicator which controls the output or not of the FCAN realization.

cc 5 : N3, the binary indicator which controls the output or not of the EXOR realization.

NOTE: These indicators (N1, N2, N3) can be used singly, or in pairs, or all three, but obviously at least one is required for some output.

2. Overlap Method Program:

cc 1 and 2 : N, the degree of the de Bruijn sequence or of the machine required.

cc 3 and 4 : NA, the number of such sequences or machines demanded; this number is limited by equation 3.4.1 (p.31).

cc 5 : N1, the binary indicator (0 or 1) which dictates whether the user wishes to include the actual de Bruijn sequence in the program output.

cc 6 : N2, the binary indicator which controls the output or not of the FCAN realization.

cc 7 : N3, the binary indicator which controls the output or not of the EXOR realization.

3. Complete Method Program:

cc 1 and 2 : NS, the degree of the de Bruijn sequence or of the machine required.

cc 3 and 4 : NA, the number of such sequences or machines desired; this number is limited by equation 2.3.2 (p.14).

cc 5 : N1, the binary indicator (0 or 1) which dictates whether or not the user wishes to include the actual de Bruijn sequence in the program output.

cc 6 : N2, the binary indicator which controls the output or not of the FCAN realization.

cc 7 : N3, the binary indicator which controls the output or not of the EXOR realization.

****

APPENDIX III: PROGRAM LISTINGS

1. SS Method: This yields one de Bruijn sequence and corresponding FSR logic , for any n.

(In the following program listing SHFT3 is another label used for the subroutine SHFTL.)

```
FORTRAN IV G LEVEL 0, MOD 0              MAIN                    DATE = 67348

0001                  DIMENSION MT(2048),MO(64),MX(1024)
0002                  EQUIVALENCE (MT(1025),MX(1))
0003                  LOGICAL CH
0004            10    READ (1,5) N,N1,N2,N3
0005             5    FORMAT (I2,3I1)
0006                  IF (N) 4,4,7
0007             7    N1=N1+1
0008                  N2=N2+1
0009                  N3=N3+1
0010                  WRITE (3,3) N
0011             3    FORMAT (/4HON= ,I2)
0012                  NM=N-1
0013                  NMT=2**(N-1)
0014                  NB=2**(31-N)
                C                         INITIALIZE STORAGE AREAS
0015                  IT=1
0016                  MT(1)=0
0017                  IO=1
0018                  MO(1)=0
0019                  IB=0
                C                         NEW WORD  ASSUME '1' BIT
0020            15    ITM=IT
0021                  IT=IT+1
0022                  MSR=MT(ITM)
0023                  CALL SHFT2(MSR)
0024                  MT(IT)=MSR+NB
0025                  IF(IB-32)1,2,2
0026             2    IB=0
0027                  IO=IO+1
0028                  MO(IO)=0
                C                         SEARCH TABLE
0029             1    DO 11 I=1,ITM
0030                  CH=MT(I).EQ.MT(IT)
0031                  IF (CH) GO TO 12
0032            11    CONTINUE
0033                  CALL SHFT1(MO(IO))
0034                  IB=IB+1
0035                  MO(IO)=MO(IO)+1
0036                  GO TO 15
                C                         '1' BIT ASSUMPTION FALSE  REMOVE IT
0037            12    MT(IT)=MT(IT)-NB
0038                  CALL SHFT1(MO(IO))
0039                  IB=IB+1
0040                  IF (MT(IT)) 15,6,15
                C                         DE BRUIJN SEQUENCE  NOW AVAILABLE
0041             6    GO TO (13,9),N1
0042             9    WRITE (3,8)(MO(I),I=1,IO)
0043             8    FORMAT (19H DE BRUIJN SEQUENCE/(1H ,2X,16Z8))
0044                  GO TO (14,13),N2
0045            14    GO TO (10,13),N3
                C                         SCAN FUNCTION  SELECT
0046            13    IP=1
0047                  DO 20 I=3,IT
0048                  MSR=MT(I)
0049                  CALL SHFT3(MSR,N)
0050                  IF (MSR) 21,20,20
0051            21    IM=I-1
0052                  IP=IP+1
0053            23    MT(IP)=MT(IM)
0054            20    CONTINUE
```

```
0055            GO TO (29,25),N2
0056      25    WRITE (3,26) (MT(I),I=1,IP)
0057      26    FORMAT (14H FCAN FUNCTION,10X,29HINCLUSIVE-OR SUMS OF PRODUCTS/(1H
                1 ,12Z10))
0058            GO TO (10,29),N3
C                                   'G' FUNCTION  REDUCTION
0059      29    IRP=0
0060            ISB=2**30
0061            DO 30 I=1,IP
0062            IF (MT(I)-ISB) 31,31,30
0063      31    IRP=IRP+1
0064      34    MT(IRP)=MT(I)
0065      30    CONTINUE
C                                   EXOR FUNCTION  GENERATE
0066            ICM=0
0067            DO 40 JM=1,NMT
0068            IXC=0
0069            ICT=NB
0070            JCT=ICT
0071            DO 41 JC=2,NMT
0072            IF (JC-JM) 35,47,50
0073      35    IF (MX(JC)) 45,45,37
C                                   BIT STRIP
0074      37    JCM=ICM
0075            DO 42 JS=1,N
0076      43    IF (JCM-JCT) 45,47,46
0077      46    CALL SHFT2 (JCM)
0078            CALL SHFT2 (JCT)
0079      42    CONTINUE
C                                   IXC  EXC-OR  COUNTER
0080      47    IF (IXC) 48,48,49
0081      48    IXC=1
0082            GO TO 45
0083      49    IXC=0
0084      45    ICT=ICT+NB
0085            JCT=ICT
0086      41    CONTINUE
C                                   FCAN FUNCTION CHECK
0087      50    DO 51 L=1,IRP
0088            IF (ICM-MT(L)) 51,53,51
0089      51    CONTINUE
0090            GO TO 56
0091      53    IF (IXC) 54,54,55
0092      54    IXC=1
0093            GO TO 56
0094      55    IXC=0
0095      56    MX(JM)=IXC
0096      57    ICM=ICM+NB
0097            JCM=ICM
0098      40    CONTINUE
C                                   EXOR TERMS GENERATE
0099            IFX=1
0100            DO 60 L=1,JM
0101            IF (MX(L)) 60,60,62
0102      62    MT(IFX)=L-1
0103            IFX=IFX+1
0104      60    CONTINUE
0105            MT(IFX)=L
0106            WRITE (3,61) (MT(I),I=1,IFX)
0107      61    FORMAT (14H EXOR FUNCTION,10X,29HEXCLUSIVE-OR SUMS OF PRODUCTS/(1H
                1 ,12Z10))
0108            GO TO 10
0109      4     STOP
0110            END
```

Sample Output:

```
N=  3
DE BRUIJN SEQUENCE
   000000E8
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
   00000000  10000000  30000000  60000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
   00000000  00000002  00000003  00000004

N=  4
DE BRUIJN SEQUENCE
   0000F650
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
   00000000  08000000  18000000  38000000  70000000  68000000
   60000000  10000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
   00000000  00000004  00000007  00000008

N=  5
DE BRUIJN SEQUENCE
   FB9AC520
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
   00000000  04000000  0C000000  10000000  30000000  78000000
   74000000  6C000000  70000000  64000000  18000000  68000000
   54000000  60000000  08000000  50000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
   00000000  00000004  00000006  00000008  0000000C  0000000E
   0000000F  00000010

N=  6
DE BRUIJN SEQUENCE
   FDE75C6D32C2A240
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
   00000000  02000000  06000000  0E000000  1E000000  3E000000
   56000000  1C000000  74000000  6A000000  55000000  70000000
   4A000000  60000000  04000000  14000000  50000000  08000000
   7C000000  7A000000  76000000  6E000000  78000000  72000000
   62000000  0C000000  1A000000  6C000000  68000000  52000000
   18000000  64000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
   00000000  00000005  00000007  00000008  0000000A  0000000B
   00000018  0000001A  0000001B  0000001C  0000001D  0000001E
   0000000C  0000000D  0000000E  00000010  00000015  00000017
   0000001F  00000020
```

N= 7
DE BRUIJN SEQUENCE
FEF9EBC76E9CB86CD5A3262C15284880

FCAN FUNCTION — INCLUSIVE-OR SUMS OF PRODUCTS

00000000	01000000	03000000	07000000	0F000000	1F000000
7C000000	79000000	73000000	67000000	1E000000	7A000000
3F000000	7E000000	70000000	78000000	77000000	6F000000
75000000	68000000	57000000	78000000	71000000	63000000
74000000	69000000	53000000	1C000000	72000000	65000000
59000000	66000000	1A000000	6A000000	55000000	56000000
62000000	45000000	60000000	02000000	0A000000	54000000
0E000000	1D000000	76000000	6D000000	5B000000	6E000000
4B000000	7C000000	61000000	06000000	0D000000	6C000000
68000000	51000000	0C000000	64000000	49000000	18000000
52000000	50000000	04000000	48000000		

EXOR FUNCTION — EXCLUSIVE-OR SUMS OF PRODUCTS

00000000	00000005	00000007	00000008	0000000A	0000000B
00000019	0000001A	0000001B	0000001C	0000001E	00000020
0000002D	0000002E	00000030	00000035	00000037	00000039
0000000C	0000000D	0000000F	00000010	00000015	00000017
00000025	00000027	00000028	0000002A	0000002B	0000002C
0000003A	0000003B	0000003C	0000003E	0000003F	00000040

N= 8
DE BRUIJN SEQUENCE
FF7E707C7B7A7978776757473727170D6C6A696866564626160551494121100

FCAN FUNCTION — INCLUSIVE-OR SUMS OF PRODUCTS

00000000	00800000	01800000	03800000	07800000	0F800000
77800000	6F800000	7E000000	7C800000	79800000	73800000
57800000	7C000000	78800000	71800000	63800000	0F000000
7A000000	74800000	69800000	53800000	1F000000	79000000
07000000	0E800000	1D800000	77000000	6E800000	76000000
55800000	57000000	74000000	68800000	51800000	0E000000
64800000	49800000	1C000000	71000000	62800000	45800000
5A800000	6B000000	56800000	6C000000	58800000	63000000
52800000	4B000000	68000000	50800000	06000000	0C800000
48800000	0C000000	62000000	44800000	18000000	61000000
51000000	0A000000	52000000	49000000	50000000	02000000
1F800000	3F800000	7F000000	7F800000	7D800000	7B800000
67800000	1F000000	7D000000	7A800000	75800000	6B800000
1F800000	73000000	76800000	6D800000	5B800000	6F000000
72800000	65800000	4B800000	78000000	70800000	61800000
6C800000	59800000	67000000	1D000000	75000000	6A800000
1C800000	73000000	66800000	4D800000	6F000000	72000000
70000000	6C800000	03000000	06800000	19000000	60000000
00000000	6A000000	54800000	53000000	1A000000	69000000
66000000	19000000	65000000	4A800000	56000000	64000000
42800000	60000000	01000000	05000000	15000000	54000000
48000000	04000000				

EXOR FUNCTION — EXCLUSIVE-OR SUMS OF PRODUCTS

00000000	00000005	00000007	00000009	0000000E	00000010
0000001D	0000001E	0000001F	00000020	00000025	00000027
00000036	00000038	0000003A	0000003C	0000003D	0000003E
00000056	00000058	00000059	0000005B	0000005C	0000005D
0000006A	00000068	0000006E	00000071	00000074	00000076
00000014	00000016	00000018	00000019	0000001B	0000001C
00000029	0000002A	0000002B	0000002E	00000031	00000034
00000045	00000047	00000049	0000004E	00000050	00000054
0000005E	0000005F	00000060	00000065	00000067	00000069
00000078	00000079	0000007C	0000007D	0000007E	00000080

2. Overlap Method: This yields any number of de Bruijn sequences and corresponding FSR logics up to a defined limit ($\leqslant 2^{n-1}$), for any n.

```
FORTRAN IV G LEVEL 0, MOD 0    MAIN                    DATE = 67348          16/47/4
0001          DIMENSION MT(8192),MQ(64),MX(4096),MR(13)
0002          EQUIVALENCE (MT(4097),MX(1)),(MQ(1),MR(1))
0003  10      READ(1,5)N,NA,N1,N2,N3
0004  5       FORMAT (2I2,3I1)
0005          IF (N) 4,4,7
0006  7       N1=N1+1
0007          N2=N2+1
0008          N3=N3+1
0009          NM=N-1
0010          NM1=2**(N-1)
0011          NB=2**(31-N)                AVAIL  IWD BITS  INTO  'MR' TABLE
      C
0012          IWD=-NB
0013          JA=0
0014  15      IF (NA-JA) 10,10,129
0015  129     JA=JA+1
0016          IWD=IWD+NB
0017          WRITE (3,3) N,IWD
0018  3       FORMAT (/4HON= ,I2,14X,13HINITIAL WORD=,Z8,12X,18H(0 Y(N) Y(N-1)..
             1,))
0019          JCM=IWD
0020          DO 11 I=1,N
0021          CALL SHFT1(JCM)
0022          IF(JCM)12,13,13
0023  12      MR(I) =0
0024          GO TO 11
0025  13      MR(I) =1
0026  11      CONTINUE                    SET UP 'P' FUNCTION IN MX TABLE
      C
0027          MT1=IWD
0028          LCM=32-N
0029          NM2=N-2
0030          CALL SHFT4(MT1,LCM)
0031          DO 14 I=1,NM1
0032          LCM=32-N
0033          ICT=I-1
0034          CALL SHFT1(ICT,LCM)
0035          IF (MT1-ICT)16,17,16
0036  17      MX(I)=MR(N)
0037          GO TO 14
0038  16      JCT=ICT
0039          JCM=MT1                     STRIP COMPARE
      C
0040          DO 18 J=1,NM2
0041          CALL SHFT2(JCT)
0042          LCM=LCM+1
0043          CALL SHFT4(JCM,LCM)
0044          IF (JCT-JCM)18,19,18
```

```
0045   19    MX(I)=MR(N-J)
0046         GO TO 14
0047   18    CONTINUE
0048         MX(I)=MR(I)
0049   14    CONTINUE
0050         LCM=32-N
0051         NWD=IWD
C                                        O/P INITIALIZED
0052         GO TO (101,102),N1
0053   102   MSR=IWD
0054         KOP=1
0055         IO=0
0056         CALL SHFTL(MSR,N)
0057         IF (MSR) 105,106,106
0058   105   LO=1
0059         GO TO 107
0060   106   LO=0
0061   107   KO=1
0062         WRITE (3,108)
0063   108   FORMAT (19H DE BRUIJN SEQUENCE)
0064   101   IT=0
C                                        GENERATE NEW WORD (NWD)
0065   125   LWD=NWD
0066         CALL SHFT2(NWD)
0067         CALL SHFTR(NWD,LCM)
0068         IFX=NWD+1
0069         MSR=MX(IFX)
0070         IF (MSR) 21,20,21
0071   20    MX(IFX)=1
0072         CALL SHFTL(NWD,LCM)
0073         GO TO 22
0074   21    MX(IFX)=0
0075         CALL SHFTL(NWD,LCM)
0076         NWD=NWD+NP
C                                        CHECK LWD FOR FCAN(STORED IN MT TABLE
0077   22    MSR=NWD
0078         CALL SHFTL(MSR,N)
0079         IF (MSR)23,24,24
0080   23    IT=IT+1
0081         MT(IT)=LWD
0082         NO=1
0083         GO TO 115
0084   24    NO=0
0085   115   IF (NWD-IWD) 133,131,133
0086   131   GO TO (111,132),N1
C                                        O/P SEQUENCE
0087   133   GO TO (125,135),N1
0088   135   IF(LO-NO)117,116,117
0089   116   KO=KO+1
0090         GO TO 125
0091   117   IF (IO-64) 118,119,119
0092   132   KOP=2
0093         MO(IO)=KO
0094   119   WRITE (3,120)(MO(I),I=1,IO)
0095   120   FORMAT(1H ,64I2)
0096         GO TO (126,111),KOP
0097   126   IO=0
0098   118   IO=IO+1
0099         MO(IO)=KO
0100         KO=1
0101         LO=NO
0102         GO TO 125
```

```
0103  111   IP=IT
0104        GO TO (29,25),N2
0105   25   WRITE (3,26) (MT(I),I=1,IP)
0106   26   FORMAT (14H ECAN FUNCTION,10X,29HINCLUSIVE-OR SUMS OF PRODUCTS,9X,
            128H(0 Y(N) Y(N-1)...Y(1) 0...0)/(1H ,12Z10))
0107        GO TO (15,29),N3
C                          'G' FUNCTION REDUCTION
0108   29   IRP=0
0109        ISB=2**30
0110        DO 30 I=1,IP
0111        IF (MT(I)-ISB) 31,31,30
0112   31   IRP=IRP+1
0113   34   MT(IRP)=MT(I)
0114   30   CONTINUE
C                          EXOR FUNCTION GENERATE
0115        ICM=0
0116        DO 40 JM=1,NMT
0117        IXC=0
0118        ICT=NB
0119        JCT=ICT
0120        DO 41 JC=2,NMT
0121        IF (JC-JM) 35,47,50
0122   35   IF (MX(JC)) 45,45,37
C                          BIT STRIP
0123   37   JCM=ICM
0124        DO 42 JS=1,N
0125   43   IF (JCM-JCT) 45,47,46
0126   46   CALL SHFT2 (JCM)
0127        CALL SHFT2 (JCT)
0128   42   CONTINUE
C                          IXC EXC-OR COUNTER
0129   47   IF (IXC) 48,48,49
0130   48   IXC=1
0131        GO TO 45
0132   49   IXC=0
0133   45   ICT=ICT+NB
0134        JCT=ICT
0135   41   CONTINUE
C                          ECAN FUNCTION CHECK
0136   50   DO 51 L=1,IRP
0137        IF (ICM-MT(L)) 51,53,51
0138   51   CONTINUE
0139        GO TO 56
0140   53   IF (IXC) 54,54,55
0141   54   IXC=1
0142        GO TO 56
0143   55   IXC=0
0144   56   MX(JM)=IXC
0145   57   ICM=ICM+NB
0146        JCM=ICM
0147   40   CONTINUE
C                          EXOR TERMS GENERATE
0148        IEX=1
0149        DO 60 L=1,JM
0150        IF (MX(L)) 60,60,62
0151   62   MT(IEX)=L-1
0152        IEX=IEX+1
0153   60   CONTINUE
0154        MT(IEX)=L
0155        WRITE (3,61) (MT(I),I=1,IEX)
0156   61   FORMAT (14H EXOR FUNCTION,10X,29HEXCLUSIVE-OR SUMS OF PRODUCTS,13
            1,26H(0...0 Y(N) Y(N-1)...Y(1))/(1H ,12Z10))
0157        GO TO 15
0158    4   STOP
0159        END
```

Sample Output:

```
N=  3                    INITIAL WORD=10000000          (0 Y(N) Y(N-1)...)
DE BRUIJN SEQUENCE
 3 1 1 3
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
  10000000  30000000  60000000  00000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
  00000000  00000002  00000003  00000004

N=  4                    INITIAL WORD=08000000          (0 Y(N) Y(N-1)...)
DE BRUIJN SEQUENCE
 4 1 2 2 1 1 1 4
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
  08000000  18000000  38000000  70000000  68000000  60000000
  10000000  00000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
  00000000  00000004  00000007  00000008

N=  5                    INITIAL WORD=00000000          (0 Y(N) Y(N-1)...)
DE BRUIJN SEQUENCE
 1 5 1 3 2 2 1 1 1 2 3 1 1 1 2 1 4
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
  (0 Y(N) Y(N-1)...Y(1) 0...0)
  74000000  6C000000  70000000  64000000  18000000  68000000
  00000000  04000000  08000000  50000000
  54000000  60000000  00000000  1C000000  3C000000  78000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
     (0...0 Y(N) Y(N-1)...Y(1))
  0000000F  00000010
  00000000  00000004  00000006  00000008  0000000C  0000000E

N=  5                    INITIAL WORD=04000000          (0 Y(N) Y(N-1)...)
DE BRUIJN SEQUENCE
 5 1 3 2 2 1 1 1 2 3 1 1 1 2 1 5
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
  (0 Y(N) Y(N-1)...Y(1) 0...0)
  6C000000  70000000  64000000  18000000  68000000  54000000
  04000000  0C000000  1C000000  3C000000  78000000  74000000
  60000000  08000000  50000000  00000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
     (0...0 Y(N) Y(N-1)...Y(1))
  0000000F  00000010
  00000000  00000004  00000006  00000008  0000000C  0000000E

N=  5                    INITIAL WORD=08000000          (0 Y(N) Y(N-1)...)
DE BRUIJN SEQUENCE
 1 5 1 2 1 1 1 1 2 3 2 1 5 2 3 1
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
  (0 Y(N) Y(N-1)...Y(1) 0...0)
  74000000  58000000  68000000  50000000  24000000  4C000000
  08000000  14000000  2C000000  5C000000  3C000000  78000000
  70000000  00000000  04000000  60000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
     (0...0 Y(N) Y(N-1)...Y(1))
  0000000B  0000000C  0000000F  00000010
  00000000  00000003  00000004  00000005  00000008  00000009
```

```
N=  6                    INITIAL WORD=00000000                    (0 Y(N) Y(N-1)...
DE BRUIJN SEQUENCE
  1 6 1 4 2 3 1 1 1 3 3 2 1 2 1 1 2 2 2 1 1 2 4 1 1 1 1 1 3 1 2 1 5
FCAN FUNCTION          INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  7C000000  7A000000  76000000  6E000000  78000000  72000000
  62000000  0C000000  1A000000  6C000000  68000000  52000000
  50000000  08000000
  00000000  02000000  06000000  0E000000  1E000000  3E000000
  66000000  1C000000  74000000  6A000000  56000000  70000000
  18000000  64000000  4A000000  60000000  04000000  14000000
EXOR FUNCTION          EXCLUSIVE-OR SUMS OF PRODUCTS
    (0...0 Y(N) Y(N-1)...Y(1))
  0000000C  0000000D  0000000E  00000010  00000015  00000017
  0000001F  00000020
  00000000  00000005  00000007  00000008  0000000A  0000000B
  00000018  0000001A  0000001B  0000001C  0000001D  0000001E

N=  6                    INITIAL WORD=02000000                    (0 Y(N) Y(N-1)...
DE BRUIJN SEQUENCE
  6 1 4 2 3 1 1 1 3 3 2 1 2 1 1 2 2 2 1 1 2 4 1 1 1 1 1 3 1 2 1 6
FCAN FUNCTION          INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  7A000000  76000000  6E000000  78000000  72000000  66000000
  0C000000  1A000000  6C000000  68000000  52000000  18000000
  08000000  00000000
  02000000  06000000  0E000000  1E000000  3E000000  7C000000
  1C000000  74000000  6A000000  56000000  70000000  62000000
  64000000  4A000000  60000000  04000000  14000000  50000000
EXOR FUNCTION          EXCLUSIVE-OR SUMS OF PRODUCTS
    (0...0 Y(N) Y(N-1)...Y(1))
  0000000C  00000000  0000000E  00000010  00000015  00000017
  0000001F  00000020
  00000000  00000005  00000007  00000008  0000000A  0000000B
  00000018  0000001A  0000001B  0000001C  0000001D  0000001E

N=  6                    INITIAL WORD=04000000                    (0 Y(N) Y(N-1)...
DE BRUIJN SEQUENCE
  1 6 1 3 1 1 1 2 1 2 2 4 2 1 1 1 1 1 2 2 1 1 3 3 3 1 2 1 6 2 4 1
FCAN FUNCTION          INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  7C000000  7A000000  76000000  5C000000  74000000  6A000000
  78000000  64000000  14000000  28000000  52000000  4C000000
  02000000  60000000
  04000000  0A000000  16000000  2E000000  5E000000  3E000000
  2C000000  5A000000  58000000  32000000  66000000  4E000000
  50000000  22000000  46000000  70000000  08000000  00000000
EXOR FUNCTION          EXCLUSIVE-OR SUMS OF PRODUCTS
    (0...0 Y(N) Y(N-1)...Y(1))
  0000000B  0000000F  00000010  00000011  00000014  00000016
  0000001F  00000020
  00000000  00000003  00000006  00000007  00000008  0000000A
  00000017  00000018  0000001A  0000001B  0000001C  0000001D

N=  6                    INITIAL WORD=06000000                    (0 Y(N) Y(N-1)...
DE BRUIJN SEQUENCE
  5 1 4 2 3 1 1 1 3 3 2 1 2 1 1 2 2 2 1 1 2 6 1 1 1 1 1 3 1 2 1 4 1
FCAN FUNCTION          INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  76000000  6E000000  78000000  72000000  66000000  1C000000
  1A000000  6C000000  68000000  52000000  18000000  64000000
  20000000  42000000
  06000000  0E000000  1E000000  3E000000  7C000000  7A000000
  74000000  6A000000  56000000  70000000  62000000  0C000000
  4A000000  00000000  04000000  14000000  50000000  08000000
EXOR FUNCTION          EXCLUSIVE-OR SUMS OF PRODUCTS
    (0...0 Y(N) Y(N-1)...Y(1))
  0000000C  0000000E  0000000F  00000012  00000014  00000015
  00000000  00000001  00000003  00000008  00000009  0000000A
  00000016  00000017  0000001B  0000001D  0000001F  00000020
```

3. Complete Method: This yields any number of de Bruijn sequences and corresponding FSR logics up to the maximum ($2^{2^{n-1}}/2^n$), for any n.

```
FORTRAN IV G LEVEL 1, MOD 0    MAIN                    DATE = 67356          01/10/5
0001          DIMENSION MT(4096),MX(4096),MF(4096),MD(64),MR(12),KRN(12),MXT(409
              16)
0002          ISR=2**30
0003    10    READ(1,5)NS,NA,N1,N2,N3
0004    5     FORMAT (2I2,3I1)
0005          IF (NS)4,4,7
0006    7     N1=N1+1
0007          N2=N2+1
0008          N3=N3+1
0009          JA=0
C                                   DEFINE REGISTERS
0010          NSM=NS-1
0011          DO 72 I=1,NSM
0012    72    MR(I)=0
C                                   INITIAL 'P' & FF FUNCTIONS
0013    300   DO 301 I=1,2
0014          MX(I)=0
0015    301   MF(I)=1
0016          MF(3) =0
0017          MF(4) =0
0018          NV=1
0019          MMK=1
C                                   SET UP 'P' FUNCTION IN MT TABLE
0020    100   NV=NV+1
0021          N=NV+1
0022          NM=N-1
0023          NMT=2**(N-1)
0024          NP=2**(31-N)
0025          IWD=0
0026          DO 251 I=1,N
0027    251   IWD=IWD+2**(31-I)
0028          IF (NV-NS+1) 253,252,252
                           ***** 2 ***** (MMK=2) *****
C
0029    252   MMK=2
0030    253   LOM=32-N
0031          NWD=IWD
0032          MRF=MR(NV)
0033          IWD=1
0034          IP=0
0035          KSF=0
0036          KSET=1
C                                   DEFINE P*(1,...,1)=0
0037          MT(NMT)=0
0038          MXT(NMT)=0
0039          NMTM=NMT-1
              NM2T=2**(N-2)
              DO 320 I=1,NMTM
              IP=IP+1
              KAL=MF(I)
              GO TO (311,313),IMP
        311   IF (I-NM2T)313,313,312
        312   IMP=2
              IP=1
        313   KAL=KAL+MX(IP)
              IF (KAL-1)315,316,315
```

```
                                                CHOOSE USING REGISTER (NV)
      C
0049  315   CALL SHFT1 (MRF)
0050        MXT(I)=-1
0051        K-SE=K-SE+1
0052        IF (KSE) 334,333,334
0053  333   KSET=2
0054        KSE=0
0055  334   IF(MRF)331,330,331
0056  330   MT(I)=0
0057        GO TO 320
0058  331   MT(I)=1
0059        GO TO 320
0060  316   IF(MF(I))317,318,317
0061  317   MT(I)=0
0062        MXT(I)=0
0063        GO TO 320
0064  318   MT(I)=1
0065        MXT(I)=1
0066  320   CONTINUE
      C                                 TRANSFER MT TO MX
0067        DO 327 I=1,NMT
0068  327   MX(I)=MT(I)
      C                                 KRN INCREMENT
0069        GO TO (113,112),KSET
0070  113   IF (KSE-31) 114,112,112
0071  114   KRN(NV)=2**(31-KSE)
0072        GO TO 322
0073  112   KRN(NV)=1
0074  322   CONTINUE
      C                                 O/P INITIALIZED
0075        GO TO (101,015),MMK
0076  15    IF (NA-JA) 10,10,129
                                        ***** 4 ***** (129,10) *****
      C
0077  129   JA=JA+1
0078        WRITE (3,3) NS,JA
0079  3     FORMAT (/4HON= ,I2,5X, 9HCASE NO.= ,I4)
0080  103   GO TO (101,102),NJ
0081  102   MSR=IWD
0082        KOP=1
0083        I0=0
0084        CALL SHFTL(MSR,N)
0085        IF (MSR) 105,106,106
0086  105   LO=1
0087        GO TO 107
0088  106   LO=0
0089  107   KO=1
0090        WRITE (3,108)
0091  108   FORMAT (19H OF BRUTJN SEQUENCE)
      C                                 GENERATE NEW WORD (NWD)
      C
0092  101   IT=0
0093        KNT=0
0094        IFX=2**N-1
0095  125   LWD=NWD
0096        IFX=IFX
0097        CALL SHFT2(NWD)
0098        CALL SHFTR(NWD,LCM)
0099        IFX=NWD+1
0100        MSR=MX(IFX)
0101        IF (MSR)-21,-20,-21
0102  20    MX(IFX)=1
0103        CALL SHFTL(NWD,LCM)
0104        IFX=IFX-1
0105        IFX=IFX*2
0106        GO TO 22
0107  21    MX(IFX)=0
0108        CALL SHFTL(NWD,LCM)
0109        NWD=NWD+NB
0110        IFX=IFX-1
0111        IFX=IFX*2+1
```

```
                                                CHECK LWD FOR FCAN,  STORE IN MT TABLE
      C
0112   22     MSR=NWD
0113          CALL SHFTL(MSR,N)
0114          IF (MSR)23,24,24
0115   23     IT=IT+1
0116          MT(IT)=LWD
0117          NO=1
0118          GO TO (180,115),MMK
0119  180     MF(LFX+1)=1
0120          GO TO 115
0121   24     NO=0
0122          GO TO (181,115),MMK
0123  181     MF(LFX+1)=0
0124  115     IF (NWD-IWD) 127,128,127
                                  ***** 1 ***** (125,100,111) *****
      C
0125  128     GO TO (100,131),MMK
0126  131     GO TO (111,132),N1
      C                                         O/P SEQUENCE
0127  127     GO TO (125,133),MMK
0128  133     GO TO (125,135),N1
0129  135     IF(LO-NO)117,116,117
0130  116     KO=KO+1
0131          GO TO 125
0132  117     IF (IO-64) 118,119,119
0133  132     KOP=2
0134          MO(IO)=KO
0135          KNT=KNT+KO
0136  119     WRITE (3,120)(MO(I),I=1,IO)
0137  120     FORMAT(1H ,64I2)
0138          GO TO (126,111),KOP
0139  126     IO=0
0140  118     IO=IO+1
0141          MO(IO)=KO
0142          KNT=KNT+KO
0143          KO=1
0144          LO=NO
0145          GO TO 125
      C                                         O/P FCAN
0146  111     IP=IT
0147          WRITE (3,28) KNT
0148   28     FORMAT (16HOSEQUENCE LENGTH,I7)
0149          GO TO (27,25),N2
0150   27     GO TO (260,29),N3
0151   25     WRITE (3,26) (MT(I),I=1,IP)
0152   26     FORMAT (14H FCAN FUNCTION,10X,29HINCLUSIVE-OR SUMS OF PRODUCTS,9X
              1,28H(0-Y(N)-Y(N-1)...Y(1)-0...0)/(1H ,12Z10))
0153          GO TO (260,29),N3
      C                                   'G' FUNCTION  REDUCTION
0154   29     IPP=0
0155          DO 30 I=1,IP
0156          IF (MT(I)-ISB) 31,31,30
0157   31     IPP=IPP+1
0158   34     MT(IPP)=MT(I)
0159   30     CONTINUE
      C                                   EXOR FUNCTION  GENERATE
0160          ICM=0
0161          DO 40 JM=1,NMT
0162          IXC=0
0163          ICT=NB
0164          JCT=ICT
0165          DO 41 JC=2,NMT
0166          IF (JC-JM) 35,47,50
0167   35     IF (MX(JC)) 45,45,37
      C                                   BIT STRIP
0168   37     JCM=ICM
0169          DO 42 JS=1,N
0170   43     IF (JCM-JCT) 45,47,46
0171   46     CALL SHFT2 (JCM)
0172          CALL SHFT2 (JCT)
0173   42     CONTINUE
```

```
      C                    IXC  EXC-OR  COUNTER
0174   47     IF (IXC) 48,48,49
0175   48     IXC=1
0176          GO TO 45
0177   49     IXC=0
0178   45     ICT=ICT+NB
0179          JCT=ICT
0180   41     CONTINUE
      C                    ECAN FUNCTION CHECK
0181   50     DO 51 L=1,IRP
0182          IF (ICM-MT(L)) 51,53,51
0183   51     CONTINUE
0184          GO TO 56
0185   53     IF (IXC) 54,54,55
0186   54     IXC=1
0187          GO TO 56
0188   55     IXC=0
0189   56     MX(JM)=IXC
0190   57     ICM=ICM+NB
0191          JCM=ICM
0192   40     CONTINUE
      C                              EXOR TERMS GENERATE
0193          IEX=1
0194          DO 60 L=1,JM
0195          IF (MX(L)) 60,60,62
0196   62     MT(IEX)=L-1
0197          IEX=IEX+1
0198   60     CONTINUE
0199          MT(IEX)=L
0200          WRITE (3,61) (MT(I),I=1,IEX)
0201   61     FORMAT (14H EXOR FUNCTION,10X,29HEXCLUSIVE-OR SUMS OF PRODUCTS,13
     1,26H(0...0 Y(N) Y(N-1)...Y(1))/(1H ,12Z10))
      C
      C                              INCREMENT REGISTER SYSTEM
0202  260     MR(NV)=MR(NV)+KRN(NV)
0203          MRE=MR(NV)
0204          IF (MRE)261,400,400
                                     ***** 3 ***** (15,300) *****
      C                              INCREMENT & TRANSFER 'MXT'
      C
0205  400     DO 420 I=1,NMT
0206          IF (MXT(I)) 413,412,412
0207   412    MX(I)=MXT(I)
0208          GO TO 420
0209   413    CALL SHFT1(MRE)
0210          IF (MRE) 416,417,416
0211   416    MX(I)=1
0212          GO TO 420
0213   417    MX(I)=0
0214   420    CONTINUE
0215          GO TO 15
0216   261    MR(NV)=0
0217          NVM=NV-1
0218          DO 264 I=1,NVM
0219          J=NV-I
0220          IF (MR(J)) 262,265,265
0221   262    MR(J)=0
0222   264    CONTINUE
0223          GO TO 10
0224   265    MR(J)=MR(J)+KRN(J)
0225          GO TO 300
0226   4      STOP
0227          END
```

Sample Output:

```
N=  3      CASE NO.=    1
DE BRUIJN SEQUENCE
 1 3 1 1 2
FCAN FUNCTION              INCLUSIVE-OR SUMS OF PRODUCTS
  00000000  20000000  50000000  30000000
EXOR FUNCTION              EXCLUSIVE-OR SUMS OF PRODUCTS
  00000000  00000001  00000003  00000004

N=  3      CASE NO.=    2
DE BRUIJN SEQUENCE
 1 1 1 3 2
FCAN FUNCTION              INCLUSIVE-OR SUMS OF PRODUCTS
  60000000  00000000  10000000  30000000
EXOR FUNCTION              EXCLUSIVE-OR SUMS OF PRODUCTS
  00000000  00000002  00000003  00000004

N=  4      CASE NO.=    1
DE BRUIJN SEQUENCE
 1 4 1 2 2 1 1 3
FCAN FUNCTION              INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  58000000  38000000
  00000000  20000000  48000000  30000000  50000000  28000000
EXOR FUNCTION              EXCLUSIVE-OR SUMS OF PRODUCTS
   (0...0 Y(N) Y(N-1)...Y(1))
  00000007  00000008
  00000000  00000001  00000002  00000003  00000005  00000006

N=  4      CASE NO.=    2
DE BRUIJN SEQUENCE
 1 1 1 2 2 4 1 3
FCAN FUNCTION              INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  58000000  38000000
  70000000  20000000  48000000  00000000  10000000  28000000
EXOR FUNCTION              EXCLUSIVE-OR SUMS OF PRODUCTS
   (0...0 Y(N) Y(N-1)...Y(1))
  00000000  00000001  00000005  00000006  00000007  00000008

N=  4      CASE NO.=    3
DE BRUIJN SEQUENCE
 1 2 2 1 1 4 1 1 3
FCAN FUNCTION              INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  58000000  38000000
  60000000  48000000  30000000  00000000  10000000  28000000
EXOR FUNCTION              EXCLUSIVE-OR SUMS OF PRODUCTS
   (0...0 Y(N) Y(N-1)...Y(1))
  00000000  00000001  00000004  00000006  00000007  00000008

N=  5      CASE NO.=    1
DE BRUIJN SEQUENCE
 1 5 1 3 2 2 1 1 1 2 3 1 1 1 2 1 4
FCAN FUNCTION              INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  24000000  40000000  38000000  68000000  54000000  58000000
  00000000  20000000  44000000  30000000  48000000  50000000
  34000000  60000000  50000000  30000000
EXOR FUNCTION              EXCLUSIVE-OR SUMS OF PRODUCTS
   (0...0 Y(N) Y(N-1)...Y(1))
  00000006  00000007  00000009  0000000B  0000000C  0000000D
  00000000  00000001  00000002  00000003  00000004  00000005
  0000000E  00000010
```

```
N=  5      CASE NO.=    2
DE BRUIJN SEQUENCE
  1 1 2 2 1 1 1 3 3 5 1 2 2 1 1 1 4
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  0C000000  00000000  10000000  24000000  18000000  68000000
  78000000  74000000  30000000  48000000  20000000  44000000
  54000000  2C000000  5C000000  38000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
     (0...0 Y(N) Y(N-1)...Y(1))
  00000000  0000000F  0000000F  00000010
  00000000  00000001  00000002  00000006  00000009  0000000B


N=  5      CASE NO.=    3
DE BRUIJN SEQUENCE
  1 2 1 1 1 3 3 1 2 5 1 2 2 1 1 1 4
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  74000000  00000000  10000000  24000000  18000000  68000000
  70000000  48000000  20000000  44000000  0C000000  38000000
  54000000  2C000000  5C000000  38000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
     (0...0 Y(N) Y(N-1)...Y(1))
  0000000C  0000000F  0000000F  00000010
  00000000  00000001  00000002  00000006  00000009  0000000B


N=  6      CASE NO.=    1
DE BRUIJN SEQUENCE
  1 6 1 4 2 3 1 1 1 3 3 2 1 2 1 1 2 2 2 1 1 2 4 1 1 1 1 1 3 1 2 1 5
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  22000000  46000000  38000000  48000000  24000000  4A000000
  4E000000  3C000000  74000000  54000000  28000000  56000000
  5E000000  3E000000
  00000000  20000000  42000000  30000000  44000000  50000000
  58000000  32000000  4C000000  68000000  52000000  26000000
  5C000000  3A000000  6C000000  5A000000  36000000  6E000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
     (0...0 Y(N) Y(N-1)...Y(1))
  00000006  00000007  00000008  00000009  0000000A  0000000B
  00000013  00000016  00000018  00000019  0000001F  00000020
  00000000  00000001  00000002  00000003  00000004  00000005
  0000000C  0000000D  0000000E  0000000F  00000011  00000012


N=  6      CASE NO.=    2
DE BRUIJN SEQUENCE
  1 1 2 3 1 1 3 2 2 1 1 4 4 6 1 2 1 3 2 2 1 1 1 1 1 2 3 1 1 1 2 1 5
FCAN FUNCTION                INCLUSIVE-OR SUMS OF PRODUCTS
 (0 Y(N) Y(N-1)...Y(1) 0...0)
  38000000  72000000  4C000000  2C000000  42000000  06000000
  64000000  14000000  28000000  52000000  26000000  10000000
  5E000000  3E000000
  7C000000  7A000000  30000000  44000000  0A000000  16000000
  0E000000  00000000  08000000  10000000  22000000  18000000
  74000000  6A000000  2C000000  5A000000  36000000  6E000000
EXOR FUNCTION                EXCLUSIVE-OR SUMS OF PRODUCTS
     (0...0 Y(N) Y(N-1)...Y(1))
  0000000B  0000000D  00000011  00000013  00000016  00000019
  00000000  00000001  00000002  00000005  00000007  0000000A
  00000014  0000001E  0000001E  00000020
```

4. Required Subroutines: This includes the listing of both the IBM 360/40 assembly language source statements and the actual machine language object code.

```
 LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT
                                    1 SHFT    START
000000                              2 SHFT1   CSECT
000000                              3         BC      15,10(15)
000000 47FF 000A            0000A   4         DC      X'5'
000004 05                           5         DC      CL5'SHFT1'
000005 E2C8C6E3F1                   6         STM     14,12,12(13)
00000A 90EC D00C            0000C   7         L       1,0(0,1)
00000E 5810 1000            00000   8         LM      2,3,0(1)
000012 9823 1000            00000   9         SLL     2,1
000016 8920 0001            00001  10         ST      2,0(1)
00001A 5021 0000            00000  11         LM      2,12,28(13)
00001E 982C D01C            0001C  12         BCR     15,14
000022 07FE                        13 SHFT2   CSECT
000028                             14         BC      15,10(15)
000028 47FF 000A            0000A  15         DC      X'5'
00002C 05                          16         DC      CL5'SHFT2'
00002D E2C8C6E3F2                  17         STM     14,12,12(13)
000032 90EC D00C            0000C  18         L       1,0(0,1)
000036 5810 1000            00000  19         LM      2,3,0(1)
00003A 9823 1000            00000  20         SLL     2,2
00003E 8920 0002            00002  21         SRL     2,1
000042 8820 0001            00001  22         ST      2,0(1)
000046 5021 0000            00000  23         LM      2,12,28(13)
00004A 982C D01C            0001C  24         BCR     15,14
00004E 07FE                        25 SHFT4   CSECT
000050                             26         BC      15,10(15)
000050 47FF 000A            0000A  27         DC      X'5'
000054 05                          28         DC      CL5'SHFT4'
000055 E2C8C6E3F4                  29         STM     14,12,12(13)
00005A 90EC D00C            0000C  30         L       2,0(0,1)
00005E 5820 1000            00000  31         L       3,4(0,1)
000062 5830 1004            00004  32         L       3,0(0,3)
000066 5830 3000            00000  33         LM      4,5,0(2)
00006A 9845 2000            00000  34         SRL     4,0(3)
00006E 8840 3000            00000  35         SLL     4,0(3)
000072 8940 3000            00000  36         ST      4,0(2)
000076 5042 0000            00000  37         LM      2,12,28(13)
00007A 982C D01C            0001C  38         BCR     15,14
00007E 07FE                        39 SHFTR   CSECT
000080                             40         BC      15,10(15)
000080 47FF 000A            0000A  41         DC      X'5'
000084 05                          42         DC      CL5'SHFTR'
000085 E2C8C6E3D9                  43         STM     14,12,12(13)
00008A 90EC D00C            0000C  44         L       2,0(0,1)
00008E 5820 1000            00000  45         L       3,4(0,1)
000092 5830 1004            00004  46         L       3,0(0,3)
000096 5830 3000            00000  47         LM      4,5,0(2)
00009A 9845 2000            00000  48         SRL     4,0(3)
00009E 8840 3000            00000  49         ST      4,0(2)
0000A2 5042 0000            00000  50         LM      2,12,28(13)
0000A6 982C D01C            0001C  51         BCR     15,14
0000AA 07FE                        52 SHFTL   CSECT
0000B0                             53         BC      15,10(15)
0000B0 47FF 000A            0000A  54         DC      X'5'
0000B4 05                          55         DC      CL5'SHFTL'
0000B5 E2C8C6E3D3                  56         STM     14,12,12(13)
0000BA 90EC D00C            0000C  57         L       2,0(0,1)
0000BE 5820 1000            00000  58         L       3,4(0,1)
0000C2 5830 1004            00004  59         L       3,0(0,3)
0000C6 5830 3000            00000  60         LM      4,5,0(2)
0000CA 9845 2000            00000  61         SLL     4,0(3)
0000CE 8940 3000            00000  62         ST      4,0(2)
0000D2 5042 0000            00000  63         LM      2,12,28(13)
0000D6 982C D01C            0001C  64         BCR     15,14
0000DA 07FE                        65         END
```

VITA

Name: John G. McEntyre

Date of birth: February 22, 1941

Place of birth: Montreal, Quebec, Canada

Education:

Primary: Ecole Garneau, Ottawa, 1954

Secondary: University of Ottawa Prep. School, 1958

B. A. Sc. (E. E.): University of Ottawa, 1965