ISSN 0002-306X. Proc. of the RA NAS and NPUA Ser. of tech. sc. 2020. V. LXXIII, N1

UDC 004.22:004.428 RADIOELECTRONICS

R.K. APIKYAN

THE LFSR PARALLEL OUTPUT BIT GENERATION

LRFS (Linear feedback shift registers) are used for generating pseudo-random sequences from 0 and 1. The sequence is called pseudo-random because of its repeating period, which means the LFSR output data is repeating in some period, however, for large repeating periods, the output bits sequences could be considered as random. An LFSR consists of a shift register where each flip-flop state is passed from its previous register and the input bit for the LFSR is a result of modulo two addition from its feedbacks. The feedbacks are the positions of registers that are used for calculating the next input values. The LFSR's output bit processing is a linear process, and for its generation, it takes more time when we are generating large sequences. As the generation process is linear, the time for generation depends on the required output bit length and the generation time can affect the overall process where the LFSR is used. Usually, LFSR are used in cryptography for encrypting and decrypting data and for signal processing in CDMA. To shorten the generation time of the LFSR output bit generation, we need to make the generation process parallel. In this article, we will discuss the methodology of the generic LFSR upcoming state definition and the parallel output generation for different LFSR with different length and feedback positions.

Keywords: LFSR, parallel generation, Java program.

Introduction. In general, if we need to execute some linear process in parallel, we need to divide it into smaller parts and execute them separately. However, we need to know the process state for each small part and start its execution from that state. The meaning of the state for LFSR is the values of its shift registers at a given moment [1]. Let's consider an LFSR with 10 shift registers and $f_1 = 2, f_2 = 3, f_3 = 4, f_4 = 6, f_5 = 7, f_6 = 10$ feedback positions, as shown in Fig. 1.

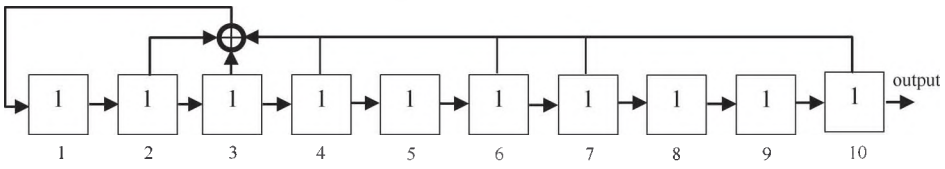

Fig. 1. An LFSR with six feedback

As we can see from Fig. 1, the LFRS state consists of ones. Let's name the first state of the LFSR as S_0 and its values are $S_0 = \{1, 1, 1, 1, 1, 1\}$. As we have mentioned earlier, the LFSR output data generation is a linear process and for

defining the next state, we need to execute an LFSR with one step. The next state of the LFSR is given in Fig. 2.

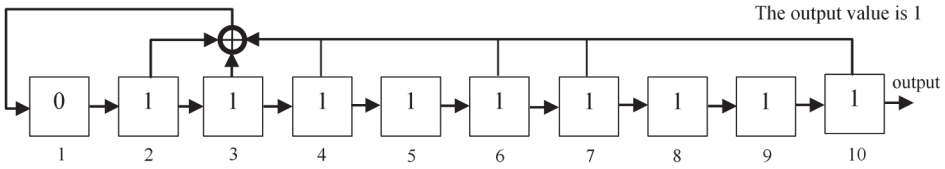

Fig. 2. The LFSR state and output bit after one time processing

As we can see from Fig. 2, the state of the LFSR $S_1 = \{ 0, 1, 1, 1, 1, 1\}$ and the output value from S_0 is *"1"*. Now, the idea becomes clearer that we can set the LFSR state and it will start the generation process from that state, and the output data will be generated from that state, but the problem here is to define the upcoming state of the LFSR in order to set it. As we can see, for starting generation from state S_n, we need to process the LFSR n times in order to get the S_n, but for parallel generation we cannot go that way. We need some formula that will allow to define the upcoming state.

From [2] we know that there is a path between the double feedback LFSR states, which means that we can apply modulo addition for the LFSR's two states and the result will be the LFSR's upcoming state. The formula is given in (1):

$$S_{i+f2} = S_i \oplus S_{i+f2-f1}, \tag{1}$$

where i is the step number, f_1 and f_2 are the LFRS's feedback positions.

However, this formula allows to define the upcoming states only for double feedback LFSR, but in our case we need a formula that allows to define the upcoming states for the generic LFSR independent of its feedback count and register length.

Methodology. For defining the path between the LFSR's states let's consider the LFSR from "Figure 1" and look at its states values for the first 10 steps. The state values for the first 10 steps are displayed in the table below.

Table

LFSR's states for first 10 processing steps

State		*State*	
0	1 1 1 1 1 1 1 1 1 1	*6*	0 0 0 1 0 0 1 1 1 1
1	0 1 1 1 1 1 1 1 1 1	*7*	1 0 0 0 1 0 0 1 1 1
2	0 0 1 1 1 1 1 1 1 1	*8*	1 1 0 0 0 1 0 0 1 1
3	1 0 0 1 1 1 1 1 1 1	*9*	1 1 1 0 0 0 1 0 0 1
4	0 1 0 0 1 1 1 1 1 1	*10*	0 1 1 1 0 0 0 1 0 0
5	0 0 1 0 0 1 1 1 1 1		

Based on discussions from [2], we know that the path between the LFSR's states could be found by applying a modulo addition to them. Here we can see the following path between the states $S_0 \oplus S_3 \oplus S_4 \oplus S_6 \oplus S_7 \oplus S_8 = S_{10}$, as shown in Fig. 3.

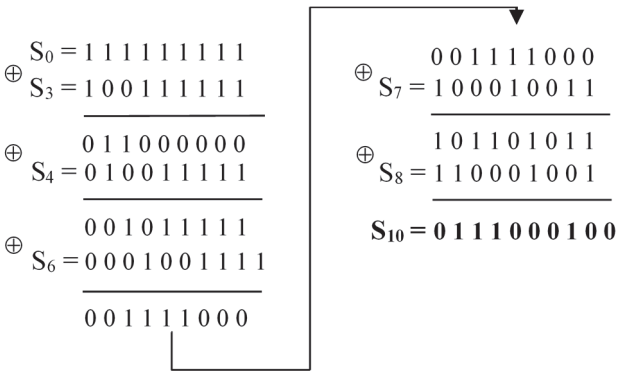

Fig. 3. The modulo addition between the LFRS's states
$S_0 \oplus S_3 \oplus S_4 \oplus S_6 \oplus S_7 \oplus S_8 = S_{10}$

This path is defined by the program from [3], which makes modulo addition of the LFSR's states and the search for the states that are equal to it.

The path that is defined for step zero is also valid for step one, which means that for step one we have the following result $S_1 \oplus S_4 \oplus S_5 \oplus S_7 \oplus S_8 \oplus S_9 = S_{11}$. We can see that whenever the step increases according to that, the indexes from the equation increase too. This allow to make the following conclusion:

$$S_i \oplus S_{i+3} \oplus S_{i+4} \oplus S_{i+6} \oplus S_{i+7} \oplus S_{i+8} = S_{i+10}, \quad (2)$$

where i is the step number.

The questions here is to understand the indexes of the states $i+3$, $i+4$, $i+5$, $i+6$, $i+7$, $i+8$, $i+10$. Let's start from the last one $i+10$. By processing program [3] for different kinds of LFSR with different lengths, we can see that number 10 from index $i+10$ is equal to the length of the LFSR. *Equation (2)* could be rewritten as follows:

$$S_i \oplus S_{i+3} \oplus S_{i+4} \oplus S_{i+6} \oplus S_{i+7} \oplus S_{i+8} = S_{i+n}, \quad (3)$$

where i is the step number, n is the length of LFSR.

For the rest of other indexes from the left side of (3) we can see that they are subtractions between the LFSR's length and the feedback position. The $i+3$ comes from $n - f_1 = 4$ where f_1 is the position of the first feedback and $f_1 = 2$. The same logic is valid for the rest of indexes. Based on this discussions we can say for the generic LFSR that it has n registers and k number feedback positions,

$$S_{i+(n-fk)} \oplus S_{i+(n-fk-1)} \oplus S_{i+(n-fk-2)} \oplus \ldots \oplus S_{i+(n-f2)} \oplus S_{i+(n-f1)} = S_{i+n}, \quad (4)$$

$$S_{i+n} = \sum_{f} S_{i+(n-f)}, \quad (5)$$

where i is the step number, n is the length of LFSR, f is the feedback position, k is the index of feedback position.

The sum symbol in (5) indicates the modulo addition. Equation (4) allows determining the upcoming states S_{i+n}, which is required for the LFSR parallel output generation.

Parallel state generation methodology. In a parallel generation program, we use the java thread pool for making the generation process parallel (Fig. 4).

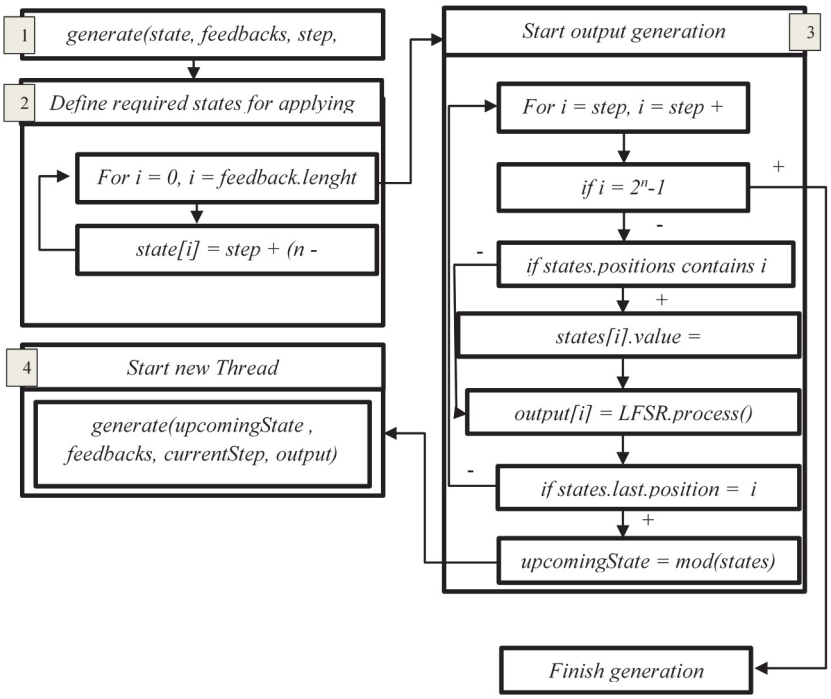

Fig. 4. Parallel Generation program block diagram

The idea here is to define a java function that will receive the LFSR state and start to generate the output bits from that state, meanwhile memorizing the states that are required for defining the upcoming state according to (5). At the point when the function determines that it has all the required states for defining the upcoming state, it starts its calculation with (5). After defining the upcoming state it will start the generation function recursively in the parallel java thread. The generation program block diagram is shown in Fig. 4, It is divided into 4 steps. Let's discuss them separately. The First state is the method structure and it shows

the input arguments. The first argument is *state*, LFSR will start its generation from this state. The second argument is *feedbacks*, which is an array of integers where each integer shows the feedback position of LFSR. The third argument is *step*, which shows the step of generation, and it is necessary to put the generated bit in the correct position inside the *output*, which is the fourth argument, and simply is an array which holds the output values of the LFSR for each generation step and has a length equal to the maximum period number of the LFSR (2^n-*1*).

In the second step of the diagram, we start the definition of the states that are required for processing (4). After the definition of the states, we start the actual LFSR output generation, which is the third step. The output generation starts from a given *step* to *step* + *n*. The first *if* statement here checks if the program has generated output data with length 2^n-*1*, which is the maximum period for the given LFSR and indicates the end of generation. Next, we check if the LFSR's current state is one of the required states that we have defined in the second step of the diagram. If it is, we save the LFSR's state value. Next, we generate the LFSR's output bit and save it in the *output* array. The generation of the output bit forces the LFSR to change its state and in the next iteration, the LFSR's state will be shifted by one bit to the left and the first bit value will be calculated based on feedback positions. In the next *if* statement we check if all states are available for performing (5), if it is true, and all states are available we process (5) and define one of the upcoming states of the LFSR and start a new generation with the calculated upcoming state in the fourth state. This generation is launched on the new thread, in hardware implementation, the new thread will be replaced by the actual LFSR.

Summary. As a result of discussions above we have equation (5) wich allows to generate the LFRS's upcoming state, and by using, that we have written the universal program that generates the output data of any kind of LFSR parallelly. Generating the output data of the LFSR parallelly is almost twice faster than doing it sequentially, which is well described in [4]. All the source codes of the LFSR parallel output generation are available in the GitHub repository [5].

REFERENCES

1. **James Bao-Yen Tsui**. Fundamentals of Global Positioning System Receivers. A Software Approach.- A Wiley intercience publication John Wiley & sons inc., New York, 2000.–255 p.
2. **Gomtsyan H.A., Apikyan R.K., Bayadyan V.H.** Double Feedback LFSR Parallel Output Generation// International Journal of Sciences: Basic and Applied Research (IJSBAR).- 2019. -Vol. 48, no. 3. -P. 143-149. link:https://www.gssrr.org/index.php/JournalOfBasicAndApplied/article/view/10308/5436
3. State comparison program's reference https://github.com/RobertApikyan/GpsGenerator/blob/parallel_and_sequential_generation/src/src/main/Main.java

4. **Гомцян О.А., Апикян Р.К.** Вычисление и измерение разницы времен при параллельной и последовательной генерации кодов C/A, Cm и Cl для Gps спутника// Электронный научный журнал “Век Качества”.- М., 2020. -N.1. -С. 158-169. - ссылка: http://www.agequal.ru/pdf/2020/120012.pdf.
5. Parallel LFSR’s programs reference https://github.com/RobertApikyan/GpsGenerator/blob/parallel_and_sequential_generation/src/src/main/ParallelLFSR.java

National Polytechnic University of Armenia. The material is received on 31.10.2019.

Ռ.Կ. ԱՊԻԿՅԱՆ

ԳԾԱՅԻՆ ՀԵՏԱԴԱՐՁ ԿԱՊՈՎ ՏԵՂԱՇԱՐԺԻ ՌԵԳԻՍՏՐՆԵՐԻ ԵԼՔԱՅԻՆ ԲԻՏԵՐԻ ԳԵՆԵՐԱՑՄԱՆ ԶՈՒԳԱՀԵՌԱՑՈՒՄԸ

Գծային հետադարձ կապով տեղաշարժման ռեգիստրները (ԳՀԿՏՌ) օգտագործվում են քվազիպատահական հաջորդականությունների գեներացման համար, որոնք բաղկացած են 1-երից և 0-ներից: Հաջորդականությունը կոչվում է քվազիպատահական, քանի որ այն ունի կրկնման պարբերություն, այնուամենայնիվ, մեծ կրկնման պարբերությունների դեպքում ելքային բիտերի հաջորդականությունը հնարավոր է դիտարկել որպես պատահական: ԳՀԿՏՌ-ն կազմված է հաջորդաբար միացված տեղափոխման ռեգիստրներից, որոնց յուրաքանչյուրի արժեքը փոխանցվում է նախորդ ռեգիստրից, իսկ մուտքային ռեգիստրի արժեքը հավասար է հետադարձ բիտերի մոդուլ երկու արժեքին: Հետադարձ բիտերի միջոցով ստացվում են մուտքային արժեքները: ԳՀԿՏՌ-ի ելքային բիտերի գեներացումը գծային է և պահանջում է երկար ժամանակ մեծ հաջորդականությունների գեներացման համար: Գեներացման ժամանակը ուղիղ համեմատական կապով կախված է պահանջվող ելքային բիտերի քանակից, և այն կարող է անդրադառնալ ընդհանուր համակարգի վրա, որտեղ օգտագործվում է ԳՀԿՏՌ-ն: Սովորաբար ԳՀԿՏՌ-ները օգտագործվում են կրիպտոգրաֆիայում՝ տվյալների կորդավորման, դեկոդավորման և CDMA-ում՝ ազդանշանի մշակման համար: Ելքային բիտերի գեներացման ժամանակի կրճատման նպատակով անհրաժեշտ է կատարել գեներացման զուգահեռացում: Աշխատանքում ներկայացվում են ԳՀԿՏՌ-ի ռեգիստրների առաջիկա արժեքների հայտնաբերման մեթոդաբանությունը և ելքային բիտերի գեներացման զուգահեռացումը ցանակացած երկարության և հետադարձ կապով ԳՀԿՏՌ-ների դեպքում:

Առանցքային բառեր. գծային հետադարձ կապով տեղաշարժման ռեգիստ (ԳՀԿՏՌ), ,զուգահեռ գեներացում, Java ծրագիր:

Р.К. АПИКЯН

ПАРАЛЛЕЛЬНАЯ ГЕНЕРАЦИЯ ВЫХОДНЫХ БИТОВ ЛИНЕЙНЫХ РЕГИСТРОВ СДВИГА С ОБРАТНОЙ СВЯЗЬЮ

Линейные регистры сдвига с обратной связью (ЛРСОС) используют для генерации псевдослучайных последовательностей, состоящих из “0” и “1”. Последовательность называется псевдослучайной из-за повторяющегося периода. Это означает, что у выходных данных ЛРСОС есть период повторения, однако для больших периодов последовательности выходных битов можно рассматривать как случайные. ЛРСОС состоят из сдвиговых регистров, где значения каждого регистра передаются от его предыдущего регистра, а входной бит для ЛРСОС является результатом сложения по модулю два из его обратных связей. Обратная связь - это позиции регистров, которые используются для расчета входных значений. Обработка выходных битов ЛРСОС является линейным процессом, и когда генерация идет для больших последовательностей, для этого требуется много времени. Поскольку процесс генерации является линейным, время генерации зависит от требуемой длины выходных битов. Время генерации может влиять на весь процесс, где используются ЛРСОС. Обычно ЛРСОС используют в криптографии для шифрования и дешифрования данных, а также для обработки сигналов в CDMA. Для сокращения времени генерации выходных битов LFSR необходимо сделать процесс генерации параллельным. В статье обсуждается методика определения предстоящего общего состояния ЛРСОС и генерации параллельного вывода для разных ЛРСОС с разной длиной и позициями обратной связи.

Ключевые слова: линейный регистр сдвига с обратной связью (ЛРСОС), параллельная генерация, программа Java.