MODELING AND SIMULATION OF CONVOLUTIONAL ENCODERS USING LOGISIM FOR TRAINING PURPOSES IN THE UNIVERSITY OF RUSE

Assist. Prof. M.Sc. Borodzhieva A. PhD.[1], M.Sc. Aliev Y.[2], Assoc. Prof. M.Sc. Ivanova G. PhD[2]
Faculty of Electrical Engineering, Electronics and Automation – University of Ruse "Angel Kanchev",
Bulgaria, Department of Telecommunications[1], Department of Computer Systems and Technologies[2]
aborodzhieva@uni-ruse.bg

Abstract: *In telecommunication, a convolutional code is a type of error-correcting code that generates parity symbols via the sliding application of a Boolean polynomial function to a data stream. A general convolutional encoder consists of a k*L-stage shift register and n modulo-2 adders, where L is the constraint length of the encoder. Convolutional codes are used to achieve reliable data transfer in numerous applications, such as digital video, radio, mobile communications and satellite communications. These codes are often implemented in concatenation with a hard-decision code, particularly Reed-Solomon codes. The material presented in the paper is used in the educational process in the University of Ruse. In order to better perception of the material active learning methods are applied. An individual assignment is given to each student and he/she has to solve the task during the practical exercise and present it at the end of the classes to the lecturer. The student should synthesize a convolutional encoder with NAND/XOR gates and flip-flops and to simulate its operation using Logisim, an educational tool for designing and simulating digital logic circuits.*

Keywords: MODELING AND SIMULATION, CONVOLUTIONAL ENCODERS, LOGISIM, ACTIVE LEARNING METHODS

1. Introduction

In telecommunications, a convolutional code is a type of error-correcting code that generates parity symbols via the sliding application of a Boolean polynomial function to a data stream. Convolutional codes were introduced in 1955 by Peter Elias. In 1967 Andrew Viterbi determined that convolutional codes could be maximum-likelihood decoded with reasonable complexity using time invariant trellis based decoders – the Viterbi algorithm [1].

A convolutional code is described by three integers, *n*, *k*, and *L*, where the ratio *r* = *k*/*n* has the same code rate significance that it has for block codes; however, *n* does not define a block or codeword length as it does for block codes. The integer *L* is a parameter known as the *constraint length*. It represents the number of *k*-tuple stages in the encoding shift register. An important characteristic of the convolutional codes is that the encoder has memory – the *n*-tuple emitted by the convolutional encoder is not only a function of an input *k*-tuple, but is also a function of the previous *L* – 1 input *k*-tuples. In practice, *n* and *k* are small integers and *L* is varied to control the capability and complexity of the code [2].

Convolutional codes are used to achieve reliable data transfer in numerous applications, such as digital video, radio, mobile communications and satellite communications. These codes are often implemented in concatenation with a hard-decision code, particularly Reed-Solomon codes. Prior to turbo codes such constructions were the most efficient, coming closest to the Shannon limit. An especially popular Viterbi-decoded convolutional code used since the Voyager program has a constraint length *L* = 7 and a rate *r* = ½. Longer constraint lengths produce more powerful codes, but the complexity of the Viterbi algorithm increases exponentially with constraint lengths, limiting these more powerful codes to deep space missions where the extra performance is easily worth the increased decoder complexity. Mars Pathfinder, Mars Exploration Rover and the Cassini probe to Saturn use *k* of 15 and a rate of 1/6; this code performs about 2 *dB* better than the simpler *L* = 7 code at a cost of 256 times in decoding complexity (compared to Voyager mission codes) [1].

2. Convolutional encoding

A typical block diagram of a digital communication system and a version of this functional diagram, focusing primarily on the convolutional encode/decode and modulate/demodulate portions of the communication link, are presented in [2]. The input message is denoted by the sequence $\mathbf{m} = m_1, m_2, \ldots, m_i, \ldots$, where each m_i represents a binary digit (bit), and i is a time index. It is assumed that each m_i is equally likely to be a one or a zero, and independent from bit to bit. Being independent, knowledge about bit m_i gives no information about m_j $(i \neq j)$. The encoder transforms each sequence **m** into a unique codeword sequence $\mathbf{U} = G(\mathbf{m})$. A key feature of convolutional codes is that a given *k*-tuple within **m** does *not* uniquely define its associated *n*-tuple within **U** since the encoding of each *k*-tuple is *not only* a function of that *k*-tuple but is also a function of $L-1$ input *k*-tuples preceding it. The sequence **U** is partitioned into a sequence of codewords $\mathbf{U} = U_1, U_2, \ldots, U_i, \ldots$. Each codeword U_i consists of binary *code symbols*, often called *channel symbols*, *channel bits* or *code bits*; unlike the input message bits the code symbols are not independent [2].

A general convolutional encoder is shown in Fig. 1. It consists of a *kL*-stage shift register and *n* modulo-2 adders, where *L* is the constraint length.

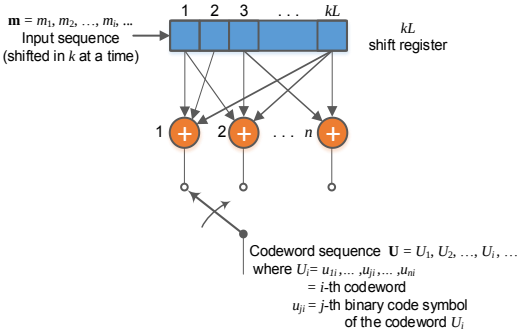

Fig. 1. *Convolutional encoder with constraint length L and rate k/n* [2]

The constraint length represents the number of *k*-bit shifts over which a single information bit can affect on the encoder's output. At each moment of time, *k* bits are shifted into the first *k* stages of the register; all bits in the register are shifted *k* stages to the right, and the outputs of the *n* adders are sequentially sampled to yield the *binary code symbols* or *code bits*. Since there are *n* code bits for each input group of *k* message bits, the code rate is *k*/*n* message bit per code bit, where *k* < *n* [2].

Only the most commonly used binary convolutional encoders for which *k* = 1 are considered in the paper. For the *k* = 1 encoder, at the i^{th} unit of time, message bit m_i is shifted into the first shift register stage, all previous bits in the register are shifted one stage to the right, and the outputs of the *n* adders are sequentially sampled and transmitted. Since there are *n* code bits for each message bit, the code rate is 1/*n*. The *n* code symbols occurring at time t_i comprise the i^{th} codeword $U_i = u_{1i}, u_{2i}, \ldots, u_{ni}$, where u_{ji} $(j = 1,2,\ldots,n)$ is the j^{th} code symbol belonging to the i^{th} codeword. For the rate 1/*n*

encoder, the *kL*-stage shift register can be referred to as a *L*-stage register, and the constraint length *L*, expressed in units of *k*-tuple stages, can be referred to as constraint length in units of bits [2].

For describing a convolutional code, the encoding function $G(\mathbf{m})$ needs to be characterized so that given an input sequence **m**, the output sequence **U** can be readily computed. Several methods are used for representing a convolutional encoder, the most popular being the *connection pictorial*, *connection vectors or polynomials*, the *state diagram*, the *tree diagram*, and the *trellis diagram*. In the paper the connection representation is used and described below.

The convolutional encoder, shown in Fig. 2, is used as a model for discussing convolutional encoders. The figure illustrates a (2, 1) convolutional encoder with constraint length $L = 3$. There are $n = 2$ modulo-2 adders; thus the code rate k/n is ½. At each moment, a bit is shifted into the leftmost stage and the bits in the register are shifted one position to the right. Next, the output switch samples the output of each modulo-2 adder (i.e., first the upper adder, then the lower adder), thus forming the code symbol pair of the codeword associated with the input bit. The sampling is repeated for each input bit. The choice of connections between the adders and the stages of the register gives rise to the characteristics of the code. Any change in the choice of connections results in a different code. The connections are not chosen or changed arbitrarily. The problem of choosing connections to yield good distance properties is complicated and has not been solved in general; however, good codes have been found by computer search for all constraint lengths less than about 20 [2].

Unlike block codes having a fixed codeword length *n*, convolutional codes have no particular block size. However, convolutional codes are often forced into a block structure by *periodic truncation*. This requires a number of zero bits to be appended to the end of the input data sequence, for the purpose of clearing or flushing the encoding shift register of the data bits. Since the added zeros carry no information, the effective code rate falls below *k*/*n*. To keep the code rate close to *k*/*n*, the truncation period is generally made as long as practical.

One way to represent the encoder is to specify a set of *n* *connection vectors*, one for each of the *n* modulo-2 adders. Each vector has dimension *L* and describes the connection of the encoding shift register to that modulo-2 adder. A one in the i^{th} position of the vector indicates that the corresponding stage in the shift register is connected to the modulo-2 adder, and a zero in a given position indicates that no connection exists between the stage and the modulo-2 adder.

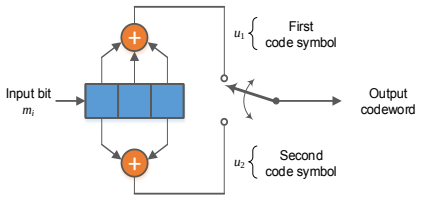

Fig.2. *Convolutional encoder (rate ½, L = 3)* [2]

For the encoder in Fig. 2, the connection vectors for the upper and for the lower connections are as follows $\mathbf{g}_1 = 111$, $\mathbf{g}_2 = 101$.

Let the message vector $\mathbf{m} = 101$ be convolutionally encoded with the encoder shown in Fig. 2. The three message bits are entered, one at a time, at times t_1, t_2, and t_3, as shown in Fig. 3. Subsequently, $(L-1) = 2$ zeros are entered at times t_4 and t_5 to flush the register and thus ensure that the tail end of the message is shifted the full length of the register. The output sequence will be 1110001011, where the leftmost symbol represents the earliest transmission. The entire output sequence, including the code symbols as a result of flushing, are needed to decode the message. Flushing the message from the encoder requires zeros, one less than the number of stages in the register, or $L-1$ flush bits. Another zero input at time t_6 is shown in Fig. 3, for verifying that the flushing is completed at time t_5. Thus, a new message can be entered at time t_6 [2].

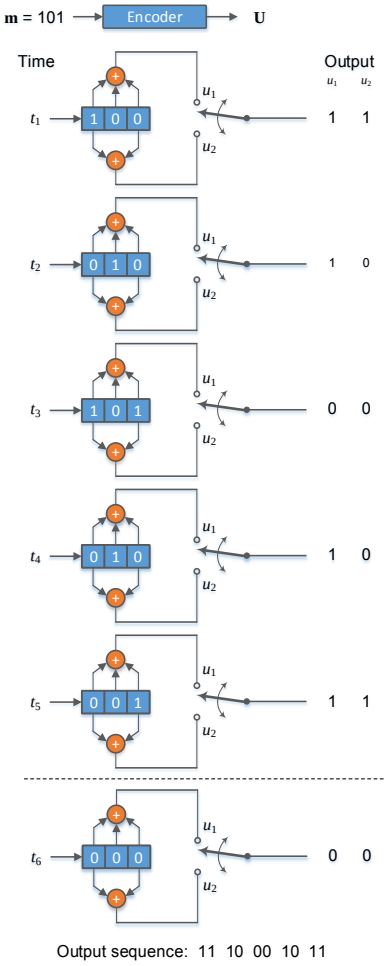

Fig. 3. *Convolutionally encoding a message sequence with a rate ½ and L = 3 encoder* [2]

The convolutional encoder can be presented in terms of its *impulse response* – the response of the encoder to a single "one" bit moving through it. The content of the register in Fig. 2, as a one moves through it, is considered below:

Register content	Codeword	
	u_1	u_2
100	1	1
010	1	0
001	1	1
Input sequences:	**1 0 0**	
Output sequences:	**11 10 11**	

The output sequence for the input "one" is called the *impulse response* of the encoder. Then, for the input sequence $\mathbf{m} = 100$, the output may be found by the *superposition* or the *linear addition* of the time-shifted input "impulses" as follows:

Input **m**	Output				
1	11	10	11		
0		00	00	00	
1			11	10	11
Modulo-2 sum:	**11**	**10**	**00**	**10**	**11**

It seems that the output is the same as that obtained in Fig. 3, demonstrating that *convolutional codes are linear*. It is from this property of generating the output by the linear addition of time-shifted impulses, or the convolution of the input sequence with the impulse response of the encoder, therefore the name *convolutional encoder* is derived.

The *effective code rate* for the example with a 3-bit input sequence and a 10-bit output sequence is $k/n = 3/10$ – quite a bit less than the rate ½ that might have been expected from knowing that each input data bit yields a pair of output channel bits. The reason for the disparity is that the final data bit into the encoder needs to be shifted through the encoder. All output channel bits are needed in the decoding process. If the message had been longer, say 300 bits, the output codeword sequence would contain 604 bits, resulting in a code rate of $300/604$ – much closer to ½ [2].

3. Logisim – an educational tool for designing and simulating digital logic circuits

Logisim is an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as the user builds them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller subcircuits, and to draw bundles of wires with a single mouse drag, Logisim can be used to design and simulate entire CPUs for educational purposes. Logisim is used by students at colleges and universities around the world in many types of classes, ranging from a brief unit on logic in general-education computer science surveys, to computer organization courses, to full-semester courses on computer architecture. The main features of the product are: 1) It is free; Logisim is open-source (GPL). 2) It runs on any machine supporting Java 5 or later; special versions are released for MacOS X and Windows. 3) The drawing interface is based on an intuitive toolbar. Color-coded wires aid in simulating and debugging a circuit. 4) The wiring tool draws horizontal and vertical wires, automatically connecting to components and to other wires. So it is very easy to draw circuits. 5) Completed circuits can be saved into a file, exported to a GIF file, or printed on a printer. 6) Circuit layouts can be used as "subcircuits" of other circuits, allowing for hierarchical circuit design. 7) Included circuit components include inputs and outputs, gates, multiplexers, arithmetic circuits, flip-flops, and RAM memory. 8) The included "combinational analysis" module allows for conversion between circuits, truth tables, and Boolean expressions [3].

Features of the components used to build the encoder

The XOR, XNOR, Even Parity, and Odd Parity gates compute the respective function of the inputs, and emit the result on the output. By default, any inputs left unconnected are ignored if the input truly has nothing attached to it, not even a wire. In this way, the user can insert a 5-input gate but only attach two inputs, and it will work as a 2-input gate; this relieves the user from having to worry about configuring the number of inputs every time he/she creates a gate. The two-input truth table for the gates is the following [4]:

x	y	XOR	XNOR	Odd	Even
0	0	0	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	1	0	1

It seems that the Odd Parity gate and the XOR gate behave identically with two inputs; similarly, the Even Parity gate and the XNOR gate behave identically. But if there are more than two specified inputs, the XOR gate will emit 1 only when there is exactly one 1 input, whereas the Odd Parity gate will emit 1 if there is an odd number of 1 inputs. The XNOR gate will emit 1 only when there is *not* exactly one 1 input, while the Even Parity gate will emit 1 if there is an even number of 1 inputs. The XOR and XNOR gates include an attribute titled Multiple-Input Behavior that allow them to be configured to use the Odd Parity and Even Parity behavior. Otherwise, it is necessary to use two 2-input XOR gates to implement a 3-input XOR gate (Fig. 4). Many authorities contend that the shaped XOR gate's behavior should correspond to the odd parity gate, but there is not agreement on this point. Logisim's default behavior for XOR gates is based on the IEEE 91 standard. It is also consistent with the intuitive meaning underlying the term *exclusive or* [4].

Each flip-flop stores a single bit of data, which is emitted through the *Q* output. Normally, the value can be controlled via the inputs. In particular, the value changes when the **clock** input, marked by a triangle on each flip-flop, rises from 0 to 1 (or otherwise as configured); on this rising edge, the value changes according to the table below [4]:

D Flip-Flop		T Flip-Flop		J-K Flip-Flop			S-R Flip-Flop		
D	*Q*	*T*	*Q*	*J*	*K*	*Q*	*S*	*R*	*Q*
0	0	0	*Q*	0	0	*Q*	0	0	*Q*
1	1	1	*Q'*	0	1	0	0	1	0
				1	0	1	1	0	1
				1	1	*Q'*	1	1	?

1) D Flip-Flop: When the clock triggers, the value remembered by the flip-flop becomes the value of the *D* (*Data*) input at that instant. **2) T Flip-Flop:** When the clock triggers, the value remembered by the flip-flop either toggles or remains the same depending on whether the *T* (*Toggle*) input is 1 or 0. **3) J-K Flip-Flop:** When the clock triggers, the value remembered by the flip-flop toggles if the *J* and *K* inputs are both 1 and the value remains the same if both are 0; if they are different, then the value becomes 1 if the *J* (*Jump*) input is 1 and 0 if the *K* (*Kill*) input is 1. **4) S-R Flip-Flop:** When the clock triggers, the value remembered by the flip-flop remains unchanged if *R* and *S* are both 0, becomes 0 if the *R* (*Reset*) input is 1, and becomes 1 if the *S* (*Set*) input is 1. The behavior in unspecified if both inputs are 1. In Logisim, the value in the flip-flop remains unchanged. By default, the clock triggers on a rising edge, i.e. when the clock input changes from 0 to 1. However, the Trigger attribute allows this to be changed to a falling edge (when the clock input changes from 1 to 0), a high level (for the duration that the clock input is 1), or a low level (for the duration that the clock input is 0). The level-trigger options are unavailable for *T* and *J-K* flip-flops, because the flip-flop behaves unpredictably when told to toggle for an indeterminate amount of time [4].

The clock toggles its output value on a regular schedule as long as ticks are enabled. A "tick" is Logisim's unit of time; the speed at which ticks occur can be selected from the Simulate menu's Tick Frequency submenu. The clock's cycle can be configured using its High Duration and Low Duration attributes. Logisim's simulation of clocks is quite unrealistic: In real circuits, multiple clocks will drift from one another and will never move in lockstep. But in Logisim, all clocks experience ticks at the same rate [4.]

The circuit of the convolutional encoder shown in Fig. 3 is drawn and tested in Logisim, to verify its operation. The circuit is built by inserting in the editing area its components – three 2-input XOR gates (XOR Gate) and three D flip-flops (D Flip-Flop) first as a sort of skeleton and then connecting them with wires. To add an input m and two outputs u1 and u2 into the diagram, the Input tool ( ) and the Output tool ( ) are selected and the pins are placed down. Then the Clock component (Clock) is placed down and connected to the flip-flops. The operation of the convolutional encoder in Fig. 3 is shown in Fig. 4. The results in Fig. 4 and its operation in Fig. 3 are identical.

4. Application in the Educational Process

The material is used in the educational process in the courses "Coding in Telecommunication Systems", "Digital Circuits" and "Pulse and Digital Devices" included in the curriculum of the specialties "Internet and Mobile Communications", "Computer Systems and Technologies", "Electronics", "Computer Management

and Automation", and "Information and Communication Technologies" for the students of the Bachelor degree in the University of Ruse.

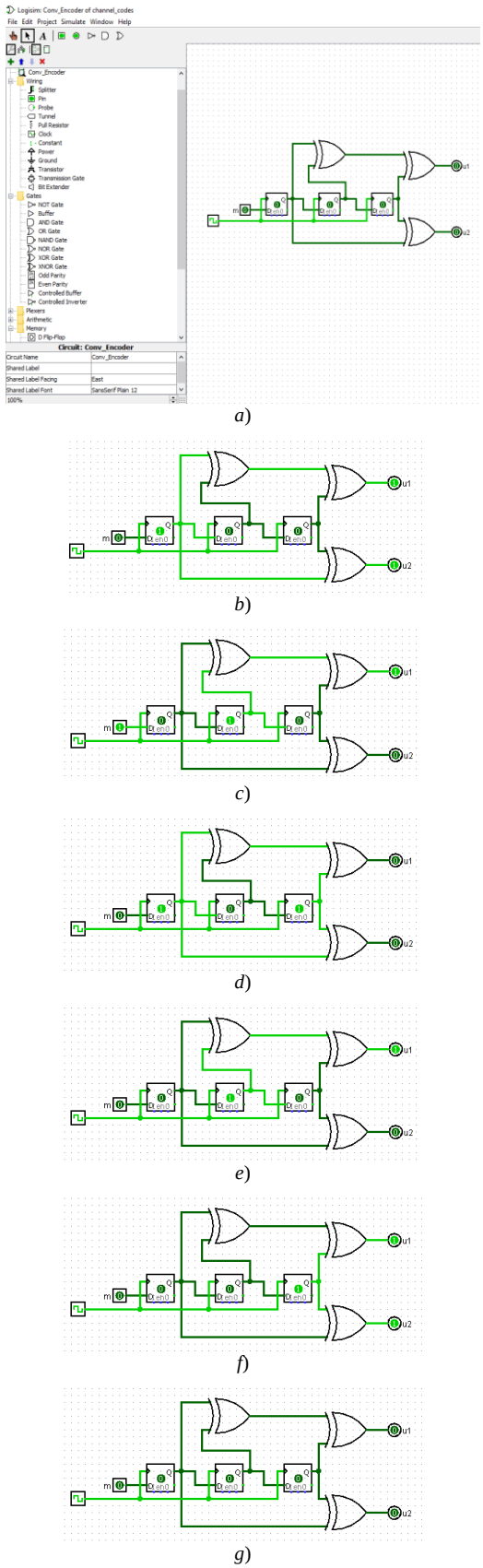

Fig. 4. a) Logisim and its simple toolbar interface + the circuit of the encoder (initial state – all-zeros) and its operations at the instances…; b) t_1; c) t_2; d) t_3; e) t_4; f) t_5; g) t_6

In order to better perception of the material active learning methods are applied in the educational process. An individual assignment is given to each student and he/she has to solve the task during the practical exercise and present it at the end of the classes to the lecturer. The student should synthesize a convolutional encoder with NAND/XOR gates and flip-flops and to simulate its operation using Logisim.

Table 1 presents the results of 10 options applied in the educational process for encoding the 3-bit message **m** = 101 using different convolutional encoders.

Table 1: *Results of 10 options applied in the educational process for encoding the 3-bit message 101 using different convolutional encoders*

№	Connection vectors for the …		Code bits
	upper adder	lower adder	
1	111	011	**10 11 01 11 11**
2	101	110	**11 01 01 01 10**
3	011	101	**01 10 10 10 11**
4	110	111	**11 11 10 11 01**
5	011	110	**01 11 11 11 10**
6	101	011	**10 01 01 01 11**
7	111	110	**11 11 01 11 10**
8	110	101	**11 10 10 10 01**
9	011	111	**01 11 10 11 11**
10	110	011	**10 11 11 11 01**

6. Conclusion

Incorporating active learning into the curriculum transforms the classroom into an exciting, dynamic learning environment. In order to easily assimilation of the material studied by students, active learning is applied. An individual assignment is given to each student. The assignment includes: 1) filling in the blanks for a given convolutional encoder (Fig. 3); 2) synthesizing a convolutional encoder with XOR gates and D flip-flops and simulating its operation using Logisim (Fig. 4); 3) synthesizing a convolutional encoder with NAND gates and D flip-flops and simulating its operation using Logisim; 4) synthesizing a convolutional encoder with XOR gates and J-K flip-flops and simulating its operation using Logisim. During the practical exercises, the student must solve his/her tasks of pre-prepared form published in the e-learning platform of the University of Ruse and submit to the teacher at the end of the classes. The opportunity for extra work is given to the curious students – for example, synthesizing a convolutional encoder with XOR gates and D flip-flops and simulating its operation using Logisim based on 4-bit or 5-bit shift registers with more than two output code bits. The material is used in the educational process in the courses "Coding in Telecommunication Systems", "Digital Circuits" and "Pulse and Digital Devices" included in the curriculum of the specialties "Internet and Mobile Communications", "Computer Systems and Technologies", "Electronics", "Computer Management and Automation", and "Information and Communication Technologies" for the students of the Bachelor degree in the University of Ruse.

The study was supported by contract of University of Ruse "Angel Kanchev", № BG05M2OP001-2.009-0011-C01, "Support for the development of human resources for research and innovation at the University of Ruse "Angel Kanchev"". The project is funded with support from the Operational Program "Science and Education for Smart Growth 2014-2020" financed by the European Social Fund of the European Union.

7. References

1. https://en.wikipedia.org/wiki/Convolutional_code (visited in November 2017).

2. Sklar, B. Digital Communications. Fundamentals and Applications. Second Edition. Communications Engineering Services, Tarzana, California, 2001. http://userspages.uob.edu.bh/-mangoud/mohab/Courses_files/sklar.pdf (visited in November 2017).

3. www.cburch.com/logisim/ (visited in November 2017).

4. Logisim Documentation (11 October 2014).