Designing Module to Perform Fast Light Block Cipher (LBC) within Microcontrollers by VHDL

Assem Badr[1*]
1 Computer department, Modern Academy for Engineering & Technology, Cairo, Egypt

ARTICLE INFO.

Article history:
Received 19 January 2022
Revised 12 March 2022
Accepted 24 March 2022
Available online 13 April 2022

Keywords:
Symmetric Encryption,
Block cipher,
Microarchitecture,
Atmel AVR,
VHDL.

ABSTRACT

Nowadays, various wireless communication sensors, detectors and controllers (such as low-end IoT) are used all over the world. They are vulnerable to the threat of hackers and attackers. Such these attacks could lead to great danger to buildings, factories, or even lives. For this reason, multi-level data encryption is highly required. But it is difficult to run a complex encryption algorithm on these embedded systems because they have limited size, power, memory, and processor. Therefore, light block ciphers (LBC) are the best solution for this case.
In this paper, a module capable of performing fast dynamic symmetric LBC (FDSLBC) will be designed based on the concept of dynamic data shuffling and exchange.
Moreover, a modification proposal within a microcontroller family by this new module. This FDSLBC module is designed by VHDL to be controlled by various proposed cipher Vector Instructions (VIs). Each one of this VI capable to carry out a complete block cipher protocol during only one clock pulse. So, security system designers can use combinations of these VIs to create fast, robust, and dynamic systems in what is called cryptographic-instructions agility.

1. Introduction

Recently, many companies and factories are used the industrial IOT wireless sensors in their control and monitoring systems (like Vibration, Temperature, Proximity, Frequency meter, Thermocouple, Pressure, AC Voltage, Air Quality Sensors)
Moreover, many peoples are used wireless sensors in their building (like Temperature, Water Detection, Humidity, Doors Open/Close Detection and Alerts). Moreover, all of them are used different types of cameras. Further, they are used different IOT wireless controllers.
All these IoT devices must be safe from data hackers. Therefore, system developers face great challenges to prevent their data from being decrypted and cracked [1]. Developers try to find powerful encryption algorithms but any complex algorithm needs complex mathematical formulas. However, the limited computational circuit in these IoT devices is not sufficient to solve complex algorithms rapidly.
The researchers have tried to solve this problem through three possible directions. The first is constructing IOT systems with lightweight ciphers through optimizing several factors in their design characteristics [2][3][4].
The second one is using compact level of the popular block ciphers like AES-128 [5] and ARIA [6].
The third, is implementing the lightweight block ciphers using either VLSI tools or the FPGA tools. Many VLSI developers modified many protocols of the light weight block cipher (such as DESL, DESXL. CURUPIRA-1, CURUPIRA-2, PUFFIN, XTEA and PRESENT) [7].
On other hand, many VHDL and FPGA developers modified many light weight cipher protocols such as crypto-processor design[8] and SEED block cipher [9].

*** Corresponding author**
E-mail address: Dr _Assem@hotmail.com

Also, many researchers have modified conventional µCs to intellectual propriety (IP) FPGA-cores using the VHDL[10]. They modified the microarchitecture (µArch) to improve their performances like the modifying of the conventional µCs-8051[11][12][13].

Like many FPGA developers and researchers, a block cipher will be implemented inside a microcontroller using the VHDL tools as will be explained later in the following sections.

Therefore, our challenge (in this research) is designing a module to perform multiple block cipher protocols with very high speed of processing (few clock pulses). Besides, the major challenge is to match this module within an up-to-date microcontroller (µC) family. According to this context, one of the most famous families of µCs called AVR was selected to further our idea.

Today, the Atmel-AVR family is one of the most famous and fastest µCs. It contains two-stages instruction pipeline (fetching & execution). Each pipeline's stage needs one clock pulse to perform its operation. Moreover, it can execute most of its instructions in one clock pulse. Its instruction-set contains about 133 RISC instructions. Each one has a 16-bit machine op-code.

From our survey, it was found that no AVR-instruction has been assigned to the last machine code 'FFFFh'. It was left as a reserved op-code as shown in the table (1). This reserved op-code will be the motivation factor (in this research) to support the AVR by new cryptographic-instructions.

Table (1) the last rows in the AVR's ISA

										AVR instructions
1	1	1	1	1	0	S	ddddd	0	bbb	BLD/BST
1	1	1	1	1	1	B	ddddd	0	bbb	SBRC/SBRS
1	1	1	1	1	X	X	ddddd	1	bbb	Reserved
X… undefined bits										

The traditional AVR is Harvard µArch type has two separate buses. The first bus relates to data memory while the other relates to program memory[14].

The traditional AVR instruction-path has width equal 16-bit to transfer the op-codes from the AVR's program memory to the AVR's instruction decoder (ID). Besides, the traditional data-path has width 8-lines to carry the temporary data (byte formats) among the ALU, the SRAM and the internal AVR modules. The most members of the AVR family have common internal SRAM characteristics. It contains 32 general-purpose-registers besides I/O registers and the extended I/O registers as shown in the figure (1) [15].

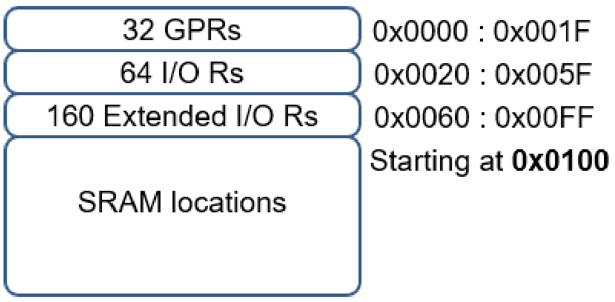


Figure (1) the organization of the AVR's SRAM

The idea and the work in this paper are organized in four sections. The first section illustrates how to modify the conventional µArch of the µC AVR to make interfacing between the proposed module (FDSLBC) and different traditional modules inside the µC. The second section is clarifying the interfacing between the FDSLBC and the SRAM inside the AVR. The third section is exposing the suggested VIs and their block-cipher protocols besides their VHDL code-samples. The fourth section demonstrates the modification behaviors by running a scenario (for encryption and decryption) includes all associated vector instructions (VIs).

2. Modifying the microarchitecture of the AVR

As mentioned earlier, the goal in this paper is to take advantage of the “FFFFh” reserved opcode to supplement the AVR with additional instructions to carry out the LBC directly. But this single reserved op-code can’t be enough to control several operations in the proposed module FDSLBC. The proposed module needs a set of op-codes to select multiple cryptographic modes. Therefore, the idea of duplication of the instruction set for 8051 was quoted [9, 10, 11].

Therefore, will modify the AVR by adding instruction switch called Instruction-Toggling-Switch (ITS) as shown in the figure (2). The dashed lines represent the added units.

When the ITS receives the machine code “FFFFh” (at any time), the next opcodes will be switched and transmitted to FDSLBC. Farther, if the ITS receives the code "FFFFh" again, it reverts back to its conventional path and transfer the subsequent op-codes to the conventional instruction decoder (ID) of the AVR, and so on.

Therefore, the added ITS has a single input bus (16-lines) from the AVR's program memory, while it has two output buses (each one 16-lines). The first bus to transfer the op-codes to the traditional ID and the second to transfer the op-codes to "FDSLBC". By default (when resetting the AVR), the ITS will be connected to the traditional ID.

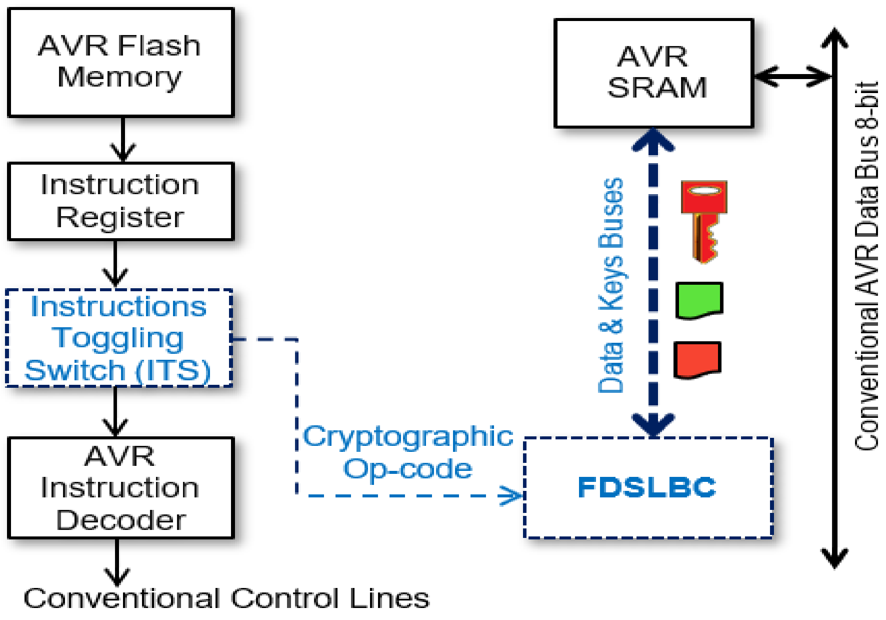


Figure (2) The additional ITS and FDSLBC in the AVR

The FDSLBC must interact with the two conventional AVR's memories (program and data memories) through two buses. The first bus to transfer the cipher keys, plain and ciphered data between the FDSLBC and the AVR’ SRAM. The second bus (op-code path) for delivering the additional op-codes from the program memory to the FDSLBC.

Since the AVR instruction-path has been modified (by ITS), the data paths will be modified too.

The FDSLBC is supplemented with register-file with sixteen byte-locations to store either plain or ciphered bytes. Using the conventional AVR data bus (8-bit) to transfer data (byte by byte) is not the optimum solution. Therefore, in order to speed-up data transfer to/from the FDSLBC, as well as reducing the SRAM's access time, the register-file (in FDSLBC) must be connected directly with its corresponding SRAM locations via dual data buses. Altogether, the FDSLBC needs three groups of buses to interface with the SRAM as shown in the figure (3).

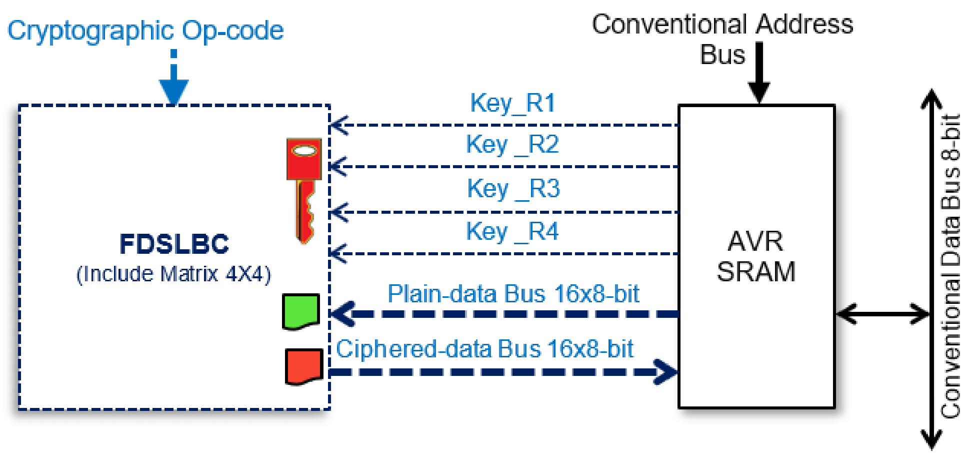


Figure (3) The interfacing between the FDSLBC and the AVR's SRAM

The first bus-group consists of 4 buses (Key_R1, Key_R2, Key_R3 and Key_R4) to transfer the proposed block cipher keys (each one 8-bit) from SRAM locations (R1, R2, R3 and R4) respectively to the FDSLBC.

The second bus-group called "Plain-data Bus" (includes 128 lines) to transfer sixteen plain bytes from the SRAM location into 4X4 matrix inside the FDSLBC. The third bus-group called "Ciphered-data Bus" (128 lines) carries sixteen ciphered bytes from the 4X4 matrix of FDSLBC to their corresponding SRAM's locations. All details of the FDSLBC operations and their VIs will be carefully explained later in the next sections.

3. Transferring and arranging data in the FDSLBC

As mentioned before in the last section, the FDSLBC contains 4X4 matrix as register-file (an array of sixteen 8-bit registers). Each cell of this matrix represented by one of 8-bit register. All 16 cells are connected concurrently with 16 locations in the SRAM. A sixteen SRAM locations is selected to be far from the special addresses (0000h to 00FFh) of the AVR. The 16 selected RAM locations will be the addresses from "0100h" to "010Fh". Each register of them will be connected directly with its corresponding SRAM location via two buses (read and write buses) as shown in the figure (5).

For instance, the RAM location ($0100h) has dual 8-bit buses "i_bus100" & "O_bus100" connected to the 1st cell (E_{11}) of the matrix as shown in the figure (4). Moreover, the last location ($010Fh) also has two buses "i_bus10F" & "O_bus10F" linked with the 16th cell (E_{44}).

As resultant, this modification allows data blocks (plain-data) to be transferred from the particular RAM locations ($0100: $010F) to FDSLBC matrix simultaneously. Furthermore, it allows data blocks (ciphered-data) to be stored from FDSLBC to its RAM locations via the cipher-bus concurrently.

$$\begin{pmatrix} E_{11} & E_{12} & E_{13} & E_{14} \\ E_{21} & E_{22} & E_{23} & E_{24} \\ E_{31} & E_{32} & E_{33} & E_{34} \\ E_{41} & E_{42} & E_{43} & E_{44} \end{pmatrix} \overset{data}{\underset{\Longleftrightarrow}{transfer}} \begin{pmatrix} \$0100 & \$0101 & \$0102 & \$0103 \\ \$0104 & \$0105 & \$0106 & \$0107 \\ \$0108 & \$0109 & \$010A & \$010B \\ \$010C & \$010D & \$010E & \$010F \end{pmatrix}$$

Figure (4) loading/retrieving 4x4 matrix from/to SRAM locations

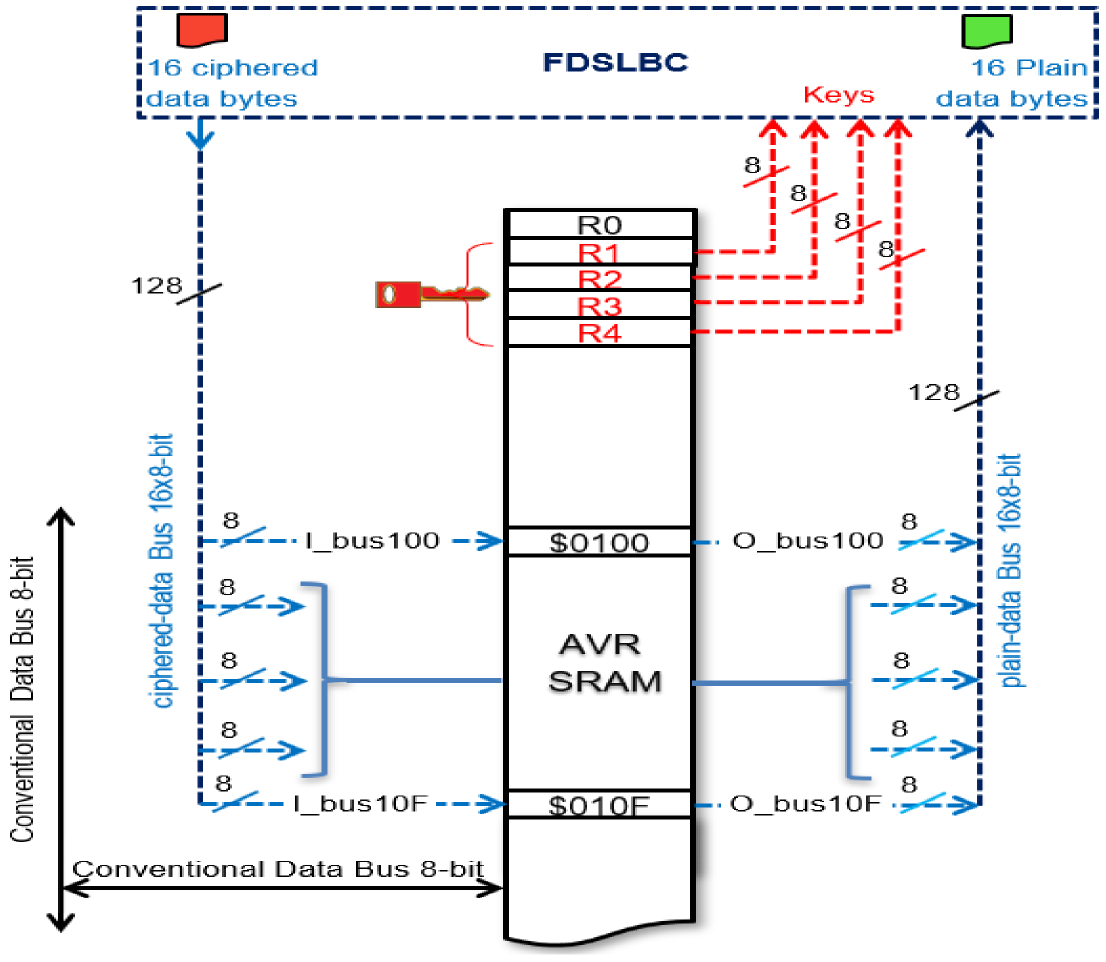


Figure (5) Modification of the traditional SRAM by additional I/O buses

4. The LBC protocols and their proposed VIs

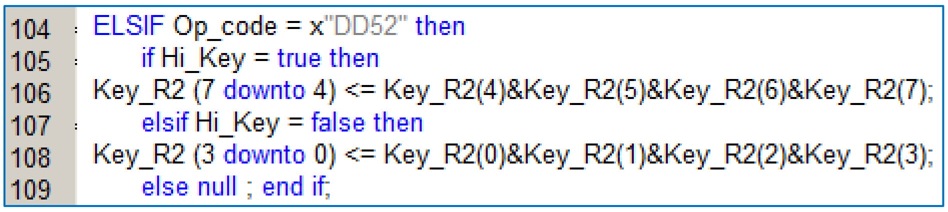
In the previous two sections, In the previous two sections, a modification has been made to the internal architecture of the AVR to accept the proposed FDSLBC module that will carry out the suggested LBC.
In this section, different LBC protocols (that will be implemented by the "FDSLBC" module) will be introduced. Besides, explain the proposed VIs that will control the FDSLBC to execute the required LBC.
For the embedded systems, the cipher algorithm is a subprogram code written by users. It consists of a set of a series of instructions that can be stored in the program memory (as a firmware).
Despite, our proposed block-cipher protocols are a firmware (machine code inside the program memory), but it has dynamic behavior because it consists of collection of cipher VIs. Each VI depends on variable cipher key(s). Therefore, any change in any cipher key will change the behavior of the total cipher algorithm as will see later.
The cipher keys are dynamic keys (changeable keys) coming from the four SRAM locations (R1, R2, R3 and R4). Users can change them based on generation of random numbers or by adequate equations or even through lock-up table. Moreover, these keys can be updated at any time depend on event(s) or under some conditions…etc. Further, users can instantly change their keys (in their codes) depending on the received data or data acknowledgments.
Furthermore, the proposed VIs doesn't rely solely on mathematical formulas (so it can be easily solved and predicted). In contrast, the new VIs will rely on the exchange, rotation, scrambling and shuffling of the data (that saved in the 4X4 byte-matrix) according to the bit states in either upper or lower nibbles of the cipher-keys.
This is mean that the four cipher-keys (from R1 to R4) will be divided into 8 nibble-keys. Users can use them in their own cipher protocols. Ultimately, at any given time, users have 8 nibbles * 4 bits leads to 2^{32} available values for his cipher protocol.

Many block cipher protocols are designed based on our idea, but only 4 protocols were chosen to prove our idea in this paper (each protocol behaves as full symmetric cipher). These proposed protocols and their VIs will be carefully explained later on.
All the proposed VIs will be executed in one clock pulse to be compatible with the execution times of the chose µC. All these VIs are listed in the table (2). It has totally 13 VIs that control the FDSLBC directly.
As mentioned earlier, the "ITS" can toggling between either the conventional instructions or new VIs. Thus, if the user needs to utilize any added VIs, he has to firstly use the VI "TOGGL" to transforming into VIs protocols. In contrast, if he needs to use the conventional instructions (at any time), he must use VI "TOGGL" again, and so on.

Table (2) List of the proposed VIs for the AVR

No.	Mnemonic	Op-code [hex]	Descriptions
1	TOGGL	FFFF	Toggling between ID and FDSLBC and vice versa at single clock
2	LOD16	DD00	Load 16 bytes from SRAM (100h: 10Fh) into the registers in FDSLBC at single clock
3	STO16	DD10	Store 16 bytes from the FDSLBC into SRAM (100h: 10Fh) at single clock
4	LDKEY	DD20	Loading eight 4-bit keys into FDSLBC at single clock
5	HIKEY	DD30	Activating the HIGH keys
6	LOWKY	DD31	Activating the LOW keys
7	CLW	DD40	Activating the clockwise data-rotation (up-rotation)
8	ACLW	DD41	Activating the Anticlockwise data-rotation (down-rotation)
9	CRY1B	DD6B	Performing data shuffling-1 (Bytes Rotation) in one clock
10	CRY2B	DD7B	Performing data shuffling-2 (Bits Rotation) in one clock
11	FLPKB	DD5B	Flip the bit arrangement of the selected nibble key from left to right or vice versa
12	CRY3B	DD8B	Performing data shuffling-3 (Bytes Shuffling) in one clock
13	CRY4B	DD9B	Performing data shuffling-4 (Up/down Bytes Rotation in 4 columns) in one clock
Note		**B** represents the upper or lower nibble of the of the selected key_R (R1, R2, R3 or R4) in the SRAM	

The second VI with mnemonic "LOD16" and op-code "DD00h" is responsible for loading 16 data-bytes concurrently (at same clock pulse) from the SRAM (starting from address 0100h) into the matrix of the FDSLBC. On the contrary, the "STO16" has op-code "DD10h" stores 16 data-bytes simultaneously from the matrix to the SRAM (starting at 0100h).
The VI "LDKEY" with op-code "DD20h" loads 4 bytes (cipher keys) concurrently from the SRAM (from address 0001h to 0004h) into the key-buffers inside the FDSLBC. The two VIs "HIKEY" and "LOWKY" that have op-codes "DD30h" and "DD31" to activate the upper and lower nibbles of the loaded key-bytes.
The two VIs "CLW" and "ACLW" that have op-codes "DD40h" and "DD41" enable the clockwise and anticlockwise data rotations respectively inside the FDSLBC.
The remaining VIs are appended with letter "B" to designate the key-number (1: 4) while the VIs (HIKEY or LOWKY) assign the nibble (low or high) inside that keys.
The set of VIs "FLPKB" consist of four VIs (FLPK1", "FLPK2", "FLPK3" and "FLPK4") which have op-codes "DD51h", "DD52", "DD53" and "DD54" respectively. It reflects (flips) the bit-order of the selected nibble-key of the selected key-byte. For instance, if the nibble-key has value "0111", then its reflection will be "1110". Code sample of the instruction "FLPKB" is illustrated in the figure (6). The line "105" checks the high nibble-key while the line "107" checks the low nibble-key. The lines "106" and "108" renders the bit reflecting for the assigned nibble-key.



```
ELSIF Op_code = x"DD52" then
    if Hi_Key = true then
Key_R2 (7 downto 4) <= Key_R2(4)&Key_R2(5)&Key_R2(6)&Key_R2(7);
    elsif Hi_Key = false then
Key_R2 (3 downto 0) <= Key_R2(0)&Key_R2(1)&Key_R2(2)&Key_R2(3);
    else null ; end if;
```

Figure (6) VHDL code of the instruction "FLPK2"

The set of VIs "CRY1B" consists of four VIs (“CRY11", " CRY12", "CRY13" and " CRY14") has op-codes "DD61h", "DD62", "DD63" and "DD64" respectively to perform the cipher protocol-1 according to data representation in the figure (7). The protocol-1 depending on VIs “CLW” or “ACLW” (to define the direction of data rotation) and the two least bit of the selected nibble (rotation enable bits). When one of the VIs “CRY1B” is executed, the bytes (in the matrix) rotate one cycle (clock or anti-clock) as the table (3).

Table (3) the different rotations of the protocol-1

The least 2 bits of Nibble	Rotation enable
00	No rotation
01	Rotation of the R H S for half-matrix
10	Rotation of the L H S for half-matrix
11	Rotation of both halves

A portion of VHDL code to implement the "CRY1B" is shown in the figure (8). The line "127" is condition of clockwise rotation. The lines from "128" to "136" indicate the concurrent bytes rotations.

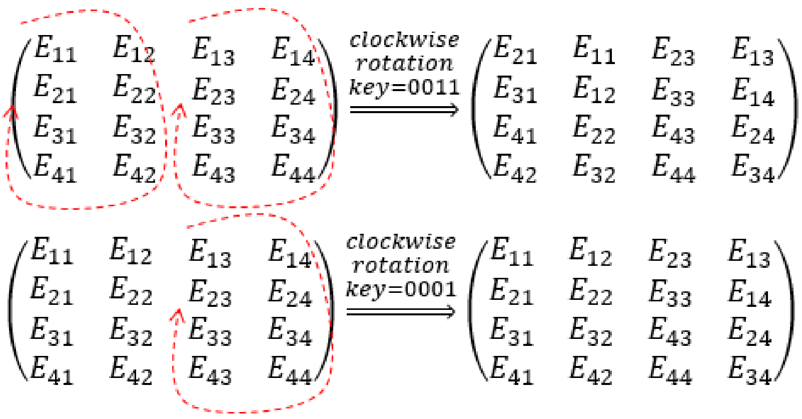
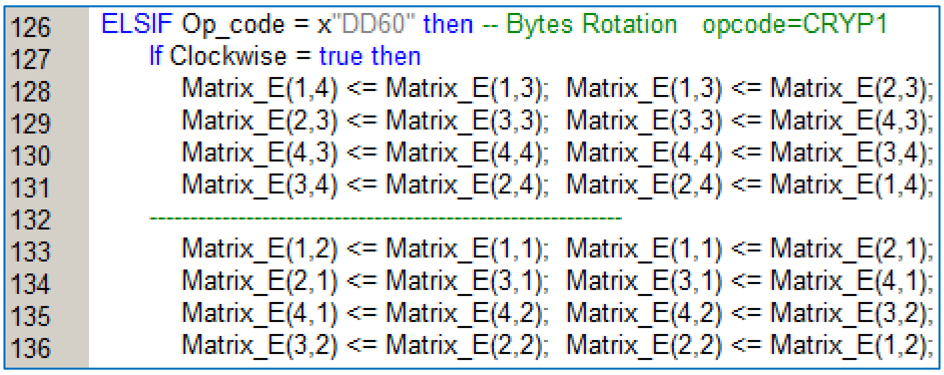


Figure (7) Bytes rotations of the Protocol-1



```
ELSIF Op_code = x"DD60" then -- Bytes Rotation   opcode=CRYP1
    If Clockwise = true then
        Matrix_E(1,4) <= Matrix_E(1,3);  Matrix_E(1,3) <= Matrix_E(2,3);
        Matrix_E(2,3) <= Matrix_E(3,3);  Matrix_E(3,3) <= Matrix_E(4,3);
        Matrix_E(4,3) <= Matrix_E(4,4);  Matrix_E(4,4) <= Matrix_E(3,4);
        Matrix_E(3,4) <= Matrix_E(2,4);  Matrix_E(2,4) <= Matrix_E(1,4);
    ------------------------------------------------------------
        Matrix_E(1,2) <= Matrix_E(1,1);  Matrix_E(1,1) <= Matrix_E(2,1);
        Matrix_E(2,1) <= Matrix_E(3,1);  Matrix_E(3,1) <= Matrix_E(4,1);
        Matrix_E(4,1) <= Matrix_E(4,2);  Matrix_E(4,2) <= Matrix_E(3,2);
        Matrix_E(3,2) <= Matrix_E(2,2);  Matrix_E(2,2) <= Matrix_E(1,2);
```

Figure (8) VHDL code of the VI "CRY1B"

The set of VIs "CRY2B" consist of four VIs ("CRY21", " CRY22", "CRY23" and " CRY24") has op-codes "DD71h", "DD72", "DD73" and "DD74" respectively. They perform bit-rotation according to data representation in the figure (9). The nibble of selected key defines the number of shifted bits in one cycle (from 1 to 7 bits) within the outer bytes $(E_{11}, E_{12}, E_{13}, E_{14}, E_{24}, E_{34})$ and $(E_{44}, E_{43}, E_{42}, E_{41}, E_{31}, E_{21})$ of the matrix. All these bytes will be configured in a register with 96-bit as shown in the VHDL-line "152" of the figure (10). The two lines "157" and "158" make clockwise bit-rotation (shifting right), while the two lines "159" and "160" make anticlockwise bit-rotation (shifting left).

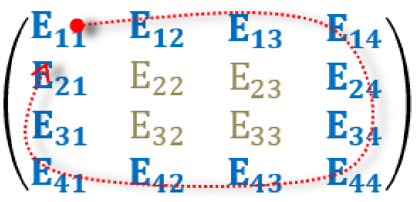
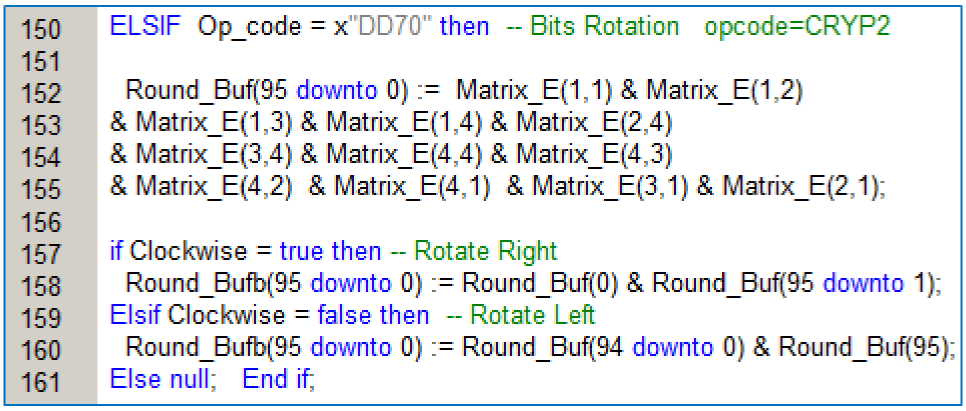


Figure (9) Bits rotations of the Protocol-2



```
150  ELSIF  Op_code = x"DD70" then  -- Bits Rotation   opcode=CRYP2
151
152    Round_Buf(95 downto 0) :=  Matrix_E(1,1) & Matrix_E(1,2)
153  & Matrix_E(1,3) & Matrix_E(1,4) & Matrix_E(2,4)
154  & Matrix_E(3,4) & Matrix_E(4,4) & Matrix_E(4,3)
155  & Matrix_E(4,2)  & Matrix_E(4,1)  & Matrix_E(3,1) & Matrix_E(2,1);
156
157  if Clockwise = true then -- Rotate Right
158    Round_Bufb(95 downto 0) := Round_Buf(0) & Round_Buf(95 downto 1);
159  Elsif Clockwise = false then  -- Rotate Left
160    Round_Bufb(95 downto 0) := Round_Buf(94 downto 0) & Round_Buf(95);
161  Else null;   End if;
```

Figure (10) VHDL code of the instruction "CRY2B"

The subsequent group of VIs "CRY3B" are ("CRY31", "CRY32", "CRY33" and "CRY34") that have op-codes "DD81h", "DD82", "DD83"and "DD84" respectively are carrying out their ciphers according to matrix multiplication in the figure (11). The 4X4 matrix represent the plain data, while the 4X1 matrix represent the selected nibble-key. The bits states of the nibble-key are either enabling or disabling their corresponding bytes-movements (shuffling indicated by arrows). For instance, if one of the VI has nibble-key equal "1001" as exposed in the figure (12), then the data-bytes in the first and fourth columns will be exchanged (swapped) only.

Subsequently, partial of its code is shown in the figure (13). The line "219" checks the activations (logic one) of both first and fourth key-bits. The line "220" is performs the data swapping between the fourth and first columns.

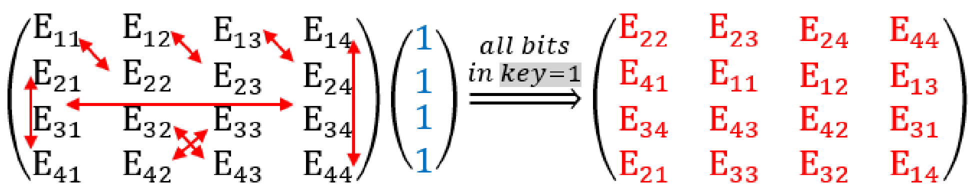


Figure (11) All bytes shuffling with the Protocol-3

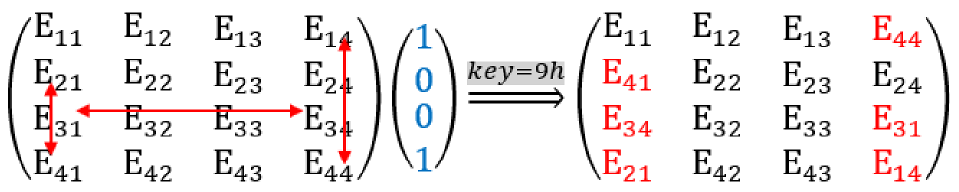
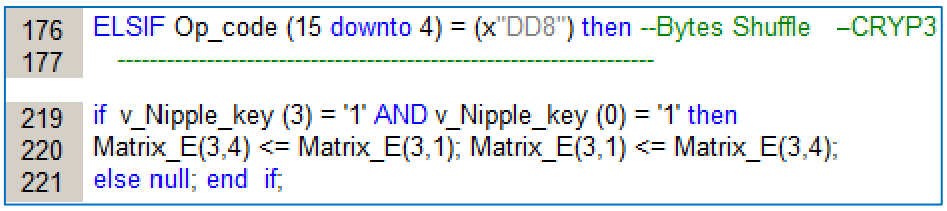


Figure (12) The selected key = "1001" on the Protocol-3



```
176  ELSIF Op_code (15 downto 4) = (x"DD8") then --Bytes Shuffle   –CRYP3
177    ------------------------------------------------------------------------

219  if  v_Nipple_key (3) = '1' AND v_Nipple_key (0) = '1' then
220  Matrix_E(3,4) <= Matrix_E(3,1); Matrix_E(3,1) <= Matrix_E(3,4);
221  else null; end  if;
```

Figure (13) segment of the code of the VI "CRY3B"

The 4th set of the VIs "CRY4B" are ("CRY41", "CRY42", "CRY43" and "CRY44") that have op-codes "DD91h", "DD92", "DD93"and "DD94" respectively to make data shuffling (protocol-4) according to the table (4).

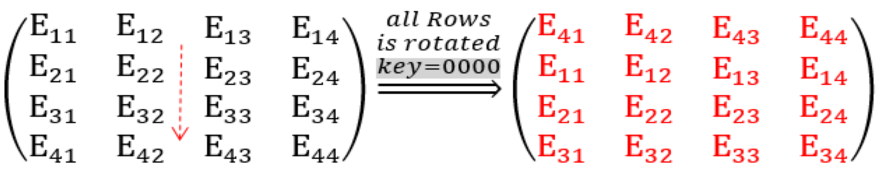


Figure (14) all rows rotation vertically of the Protocol-4

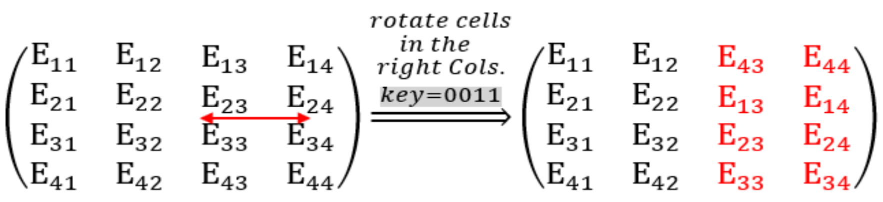


Figure (15) the 2 right columns are exchanged of the Protocol-4

Table (4) data shuffling of the protocol-4

Nibble	One cycle data shuffle	Nibble	One cycle data shuffle
0011	Bytes in the two right columns rotate	0111	Bytes in 2nd & 3rd & 4th columns rotate
1100	Bytes in the two left columns rotate	1011	Bytes in 1st & 3rd & 4th columns rotate
0110	Middle columns replacing	1101	Bytes in 1st & 2nd & 4th columns rotate
1001	outer columns replacing	1110	Bytes in 1st & 2nd & 3rd columns rotate
1010	1st & 3rd cols. replacing	1111	All columns rotate
0101	2nd & 4th cols. replacing	0001	Bytes in 1st & 2nd & 3rd rows rotate
0000	All rows rotate	0010	Bytes in 1st & 2nd & 4th rows rotate
1000	Bytes 2nd & 3rd & 4th rows rotate	0100	Bytes in 1st & 3rd & 4th rows rotate

For instance, if the VI "CRY4B" has nibble-key equal “0000”, all rows rotate vertically as seen in the figure (14). When the nibble-key equal "0011", then the data-bytes in the fourth and third columns are replaced as illustrated in the figure (17).
Eventually, all VIs are designed without operands to reducing their fetching times. Each one of them is executed in single clock pulse. In the following section will use our VIs to perform short scenario for data encryption and decryption using the module FDSLBC.

5. Running the protocols of the LBC

In this section, we will check our proposed module to ensure that it meets the proposed specifications and achieves the intended goal.
As illustrated in the table (5), there are three categories of VIs can check them. The first category called the "Block-Transfer" for transferring blocks of plain, keys and ciphered data. The second category called "Cipher-Attributes" for adding the desired features to the cipher protocols. The last category "Cipher Protocols" to perform the different ciphers protocols.

Table (5) The three categories of the proposed VIs

Cipher-Attributes	Cipher Protocols	Block-Transfer
HIKEY/ LOWKY	CRY1B	LOD16
CLW/ ACLW	CRY2B	STO16
FLPKB	CRY3B	LDKEY
	CRY4B	

As mentioned before, this modification carries out LBC protocols for embedded systems, consequently the proposed FDSLBC designed by VHDL codes. Moreover, the behavior of this VHDL code will be displayed on the reliable simulator "Modelsim" during the different VIs executions.
A scenario will be assumed to illustrate our idea by checking all VIs mentioned in this paper. This scenario includes two phases (encoding and decoding stages) as shown in figure (16). The first phase indicates an encryption algorithm while the second phase indicates algorithm of the compatible decryption. Each phase

has 13 steps of different VIs. A set of plain data was assumed with 16 bytes (128 bits) as the figure (17). The assumed scenario will depend on only two cipher-keys (key_R1 = E3h) with nibble-keys ($B1_H = Eh \text{ and } B1_L = 3h$) and (key_R2 = 71h) has nibble-keys ($B2_H = 7h \text{ and } B2_L = 1h$).

$$\begin{pmatrix} E_{11} & E_{12} & E_{13} & E_{14} \\ E_{21} & E_{22} & E_{23} & E_{24} \\ E_{31} & E_{32} & E_{33} & E_{34} \\ E_{41} & E_{42} & E_{43} & E_{44} \end{pmatrix} = \begin{pmatrix} 11h & 12h & 13h & 14h \\ 21h & 22h & 23h & 24h \\ 31h & 32h & 33h & 34h \\ 41h & 42h & 43h & 44h \end{pmatrix}$$

Figure (17) The plain data inside the 4X4 matrix

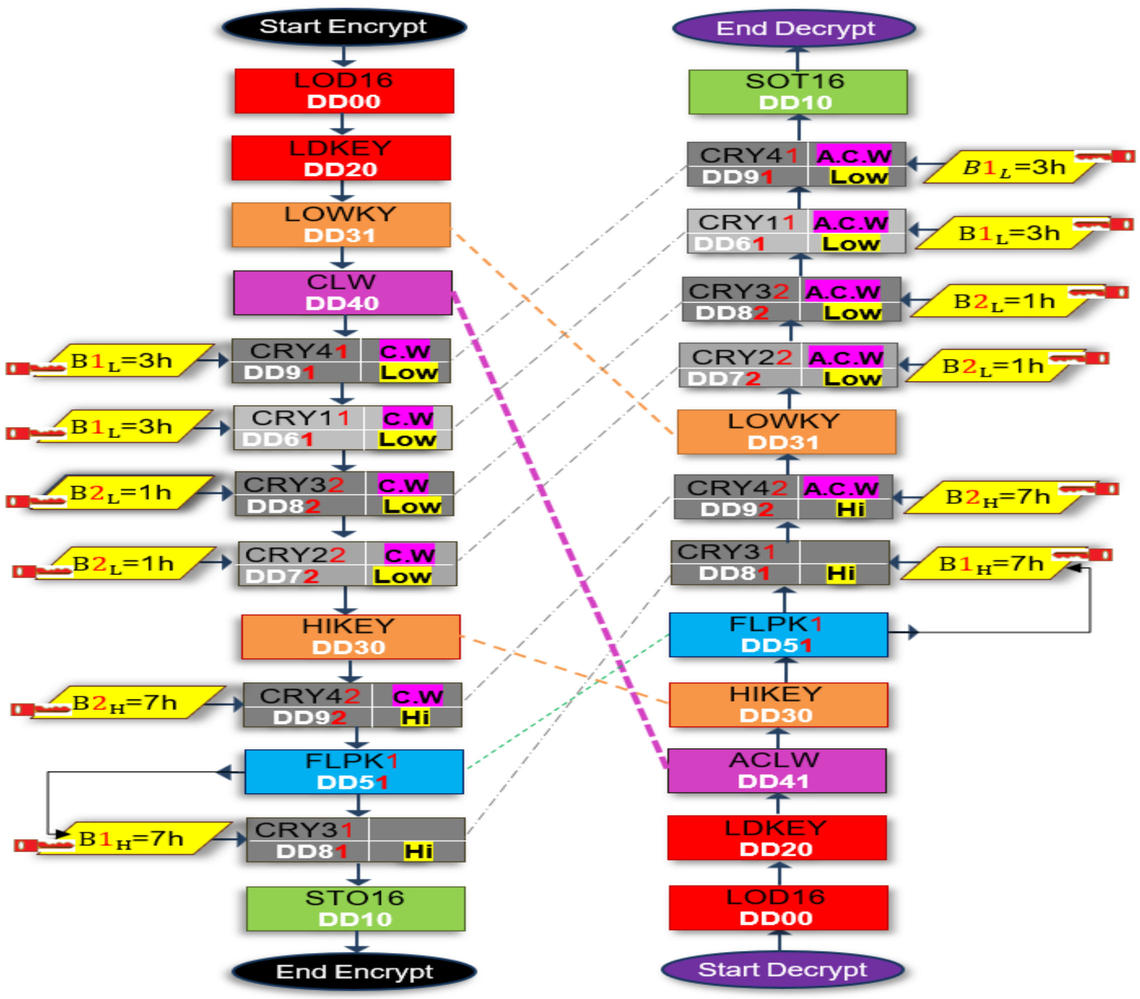


Figure (16) Scenario of two LBC algorithm, the 1st for encryption phase the 2nd for decryption phase

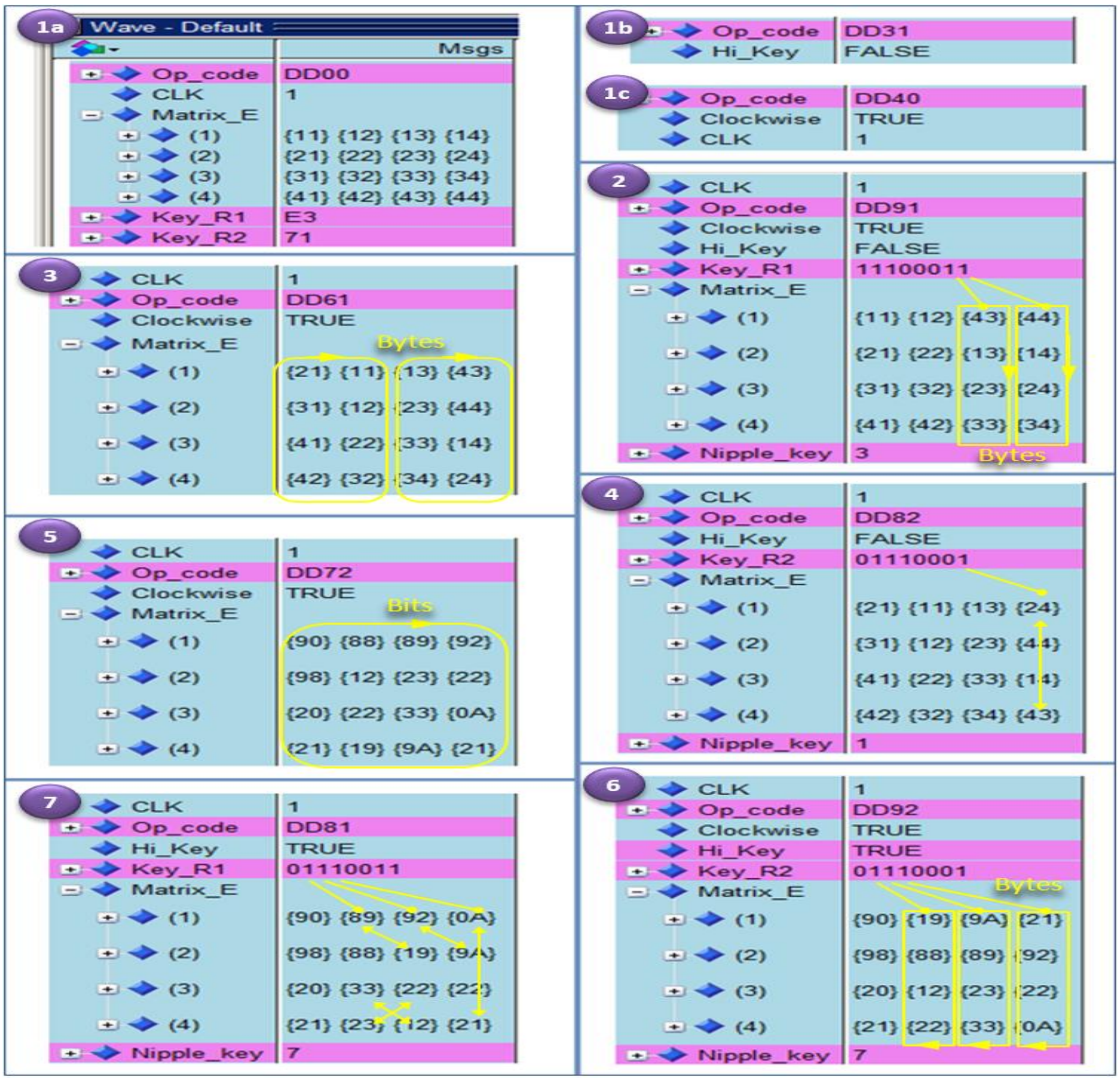
All steps of our scenario will now be discussed guided by the figure (16) and present its results in figure (18) and figure (19).

The encryption process begins by using VI "LOD16" (DD00h) to load the assumed plain data (16 bytes) from the SRAM to the 4X4 matrix as seen in the first step of the figure (16) and step “1a” of the figure (18). The 2nd VI is "LDKEY" (DD20h) loads the cipher-keys from SRAM as shown in the step "1a" of the figure (18). The 3rd VI "LOWKY" (DD31h) enables the FDSLBC to handle only the lower nibble-keys as illustrated in the step "1b" of the same figure. The 4th VI "CLW" (DD40h) activates the clockwise rotations inside the FDSLBC as shown in step "1c". The 5th VI "CRY41" (DD91H) to perform the Protocol-4. The least significant digit (LSD) of this VI is “1”, this mean that the first cipher-key is assigned. In this moment, the activated lower nibble-key ($B1_L = 0011$) is the operational key. According to the table (4), the $B1_L =$

0011 Leeds to rotation of the data bytes of the two right hand columns of the matrix as seen in step "2" of figure (18). The 6th VI "CRY11" (DD61h) performs the Protocol-1. The lowers nibbles of cipher-keys are still activated ($B1_L = 0011$); therefore, the bytes rotate as shown in the 4th row of table (3) and step “3” of figure (18).

The 7th VI "CRY32" (DD82h) performs the Protocol-3. It depends on the lower nibble ($B2_L = 0001$) of the key-2 (the LSD =2). So, only the data-bytes in the right column of the matrix will be transferred (guided with figure (12)) and as shown in step "4" of figure (18). The eighth VI "CRY22" (DD72h) performs the Protocol-2. It rotates one bite of all outer bytes inside the matrix as indicated in the figure (9) and step "5" of figure (18). The 9th VI "HIKEY" (DD30h) enables the FDSLBC to deal with the upper nibble-keys only.

The 10th VI "CRY42" (DD92h) performs the Protocol-4. It depends on the higher nibble ($B2_H = 0111$) of key-2 (LSD = 2). Thus, the bytes in the three right columns of the matrix rotate (as 1st row of table (4)) in the clockwise direction as demonstrated in step "6" of figure (18). The 11th VI "FLPK1" reflects the bit-order of the current nibble-key. In our scenario, the upper nibble-key-1 equal "1110", so its reflection will be "0111". The twelfth VI "CRY31" performs the Protocol-3. It relies on the reflected upper nibble of the key-1 (LSD =1). The reflected nibble-key has code "1110", thus it allows the data-bytes to transfer among the three right columns of the matrix as clarified in the step "7" of figure (18). The first phase is ended by the thirteenth VI "STO16" to load 16 data bytes (represent a ciphered text) from the FDSLBC to the SRAM locations.



```
Wave - Default
                 Msgs
Op_code          DD00
CLK              1
Matrix_E
  (1)            {11} {12} {13} {14}
  (2)            {21} {22} {23} {24}
  (3)            {31} {32} {33} {34}
  (4)            {41} {42} {43} {44}
Key_R1           E3
Key_R2           71
```

```
Op_code          DD31
Hi_Key           FALSE
```

```
Op_code          DD40
Clockwise        TRUE
CLK              1
```

```
CLK              1
Op_code          DD91
Clockwise        TRUE
Hi_Key           FALSE
Key_R1           11100011
Matrix_E
  (1)            {11} {12} {43} {44}
  (2)            {21} {22} {13} {14}
  (3)            {31} {32} {23} {24}
  (4)            {41} {42} {33} {34}
Nipple_key       3
```

```
CLK              1
Op_code          DD61
Clockwise        TRUE
Matrix_E
  (1)            {21} {11} {13} {43}
  (2)            {31} {12} {23} {44}
  (3)            {41} {22} {33} {14}
  (4)            {42} {32} {34} {24}
```

```
CLK              1
Op_code          DD82
Hi_Key           FALSE
Key_R2           01110001
Matrix_E
  (1)            {21} {11} {13} {24}
  (2)            {31} {12} {23} {44}
  (3)            {41} {22} {33} {14}
  (4)            {42} {32} {34} {43}
Nipple_key       1
```

```
CLK              1
Op_code          DD72
Clockwise        TRUE
Matrix_E
  (1)            {90} {88} {89} {92}
  (2)            {98} {12} {23} {22}
  (3)            {20} {22} {33} {0A}
  (4)            {21} {19} {9A} {21}
```

```
CLK              1
Op_code          DD92
Clockwise        TRUE
Hi_Key           TRUE
Key_R2           01110001
Matrix_E
  (1)            {90} {19} {9A} {21}
  (2)            {98} {88} {89} {92}
  (3)            {20} {12} {23} {22}
  (4)            {21} {22} {33} {0A}
Nipple_key       7
```

```
CLK              1
Op_code          DD81
Hi_Key           TRUE
Key_R1           01110011
Matrix_E
  (1)            {90} {89} {92} {0A}
  (2)            {98} {88} {19} {9A}
  (3)            {20} {33} {22} {22}
  (4)            {21} {23} {12} {21}
Nipple_key       7
```

Figure (18) snap shots of the encryption steps

In the symmetric key encryption, the decryption algorithm is serving as a mirror of its encryption algorithm. All the used VIs will be repeated with their same attributes but in the opposite direction as demonstrated in figure (19).

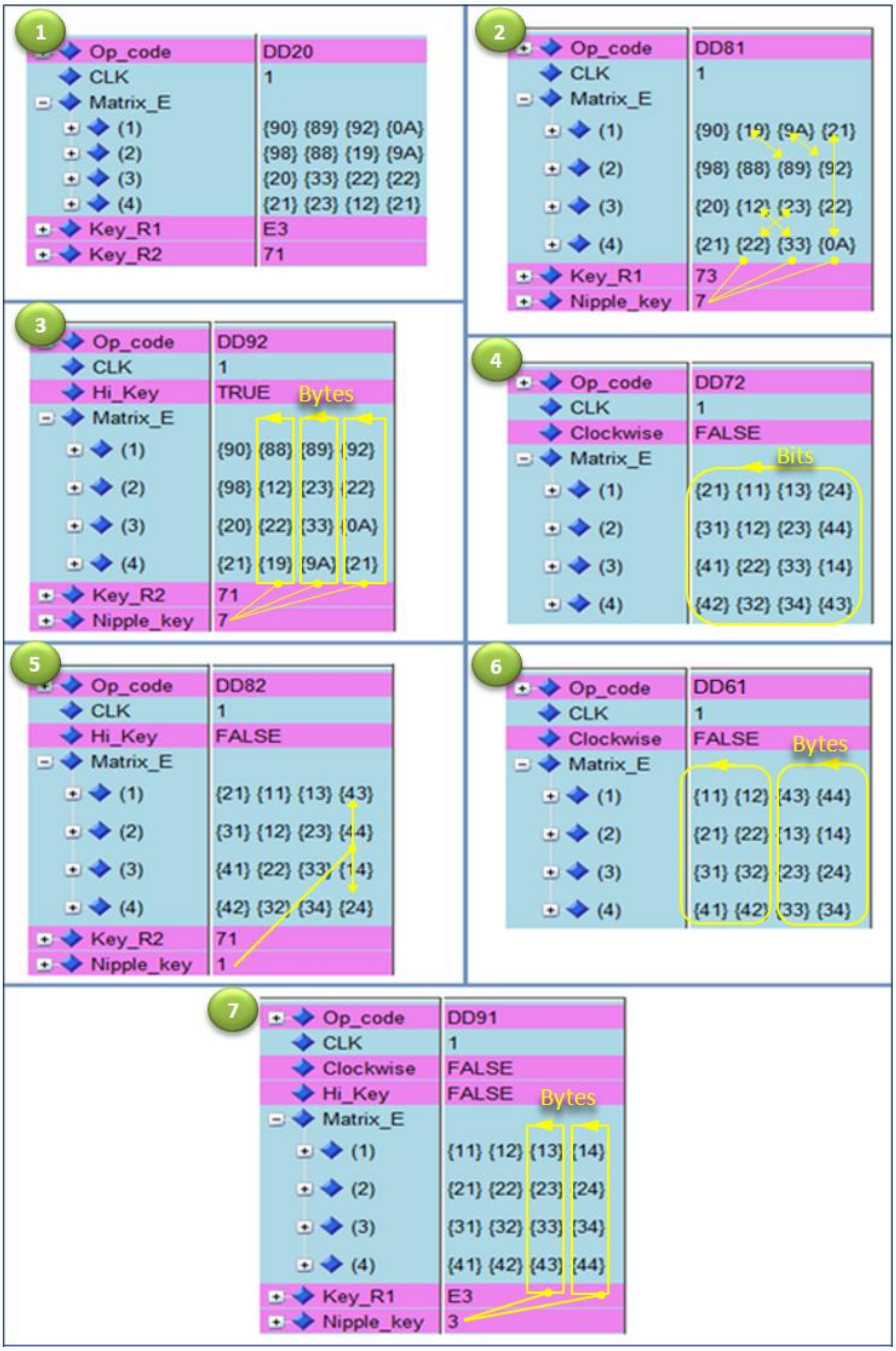
The decryption algorithm begins by loading 16 data bytes (represent a cipher text or section of a great cipher text) from the SRAM into the FDSLBC using the VI "LOD16" as illustrated in the step "1" of the figure (19). It is clear that the loaded 16 data-bytes are similar to the early ciphered from the previous encryption algorithm. The next VI "LDKEY" loads the same cipher keys that loaded for the mentioned encryption process (symmetric keys).

Previously, the data in the encryption phase rotated in clockwise directions, therefore all the data in this decryption phase would rotate in the opposite direction (anticlockwise). Therefore, the third VI "ACLW" permits the FDSLBC to rotate all data anticlockwise.

The last encryption protocol was Protocol-3 with reflected high nibble-key. Thus, the fourth VI will be "HIKEY" to enable the upper keys. Moreover, the fifth VI is "FLPK1" to reflect the nibble-key number one. The sixth VI "CRY31" is used to carry out the Protocol-3. According to the reflected nibble-key "1" with code "1110", the data-bytes are moving among the least three right columns as displayed in step "2" of the figure (19).

The seventh VI "CRY42" used to carry out the Protocol-4. It depends on the upper nibble of the second key (LSD=2) that has code equal "0111", so the least three columns will rotate vertically anticlockwise as demonstrated in step "3". The eighths VI "LOWKY" used to handle all lower nibbles of all loaded keys. The ninth VI "CRY2B" used to carry out the Protocol-2. It rotates anticlockwise all bits in the exterior bytes of the matrix as in the step "4". The tenth VI "CRY32" carries out the Protocol-3. It depends on the lower nibble of the second key. The lower nibble has code "0001", so the data-bytes transfers via the least column only as shown in step "5". The eleventh VI "CRY1B" carries out the Protocol-1, so it rotates all bytes anticlockwise for both left and right halves of matrix together as shown in step "6". The 12th VI "CRY41" carries out the Protocol-4. It depends on the nibble-key-1. It has code "0011". Therefore, all bytes in the two least columns of the matrix rotate vertically in the direction anticlockwise as demonstrated in step "7". The 13th VI "STO16" stores the original data into the SRAM.

As a resultant, it can be realized that the result of the decryption algorithm that shown in the figure (19) has completely restored each original plaintext in their original arrangements that were encoded in the encryption stage as shown in figure (18).



Op_code	DD20
CLK	1
Matrix_E	
(1)	{90} {89} {92} {0A}
(2)	{98} {88} {19} {9A}
(3)	{20} {33} {22} {22}
(4)	{21} {23} {12} {21}
Key_R1	E3
Key_R2	71

Op_code	DD81
CLK	1
Matrix_E	
(1)	{90} {19} {9A} {21}
(2)	{98} {88} {89} {92}
(3)	{20} {12} {23} {22}
(4)	{21} {22} {33} {0A}
Key_R1	73
Nipple_key	7

Op_code	DD92
CLK	1
Hi_Key	TRUE
Matrix_E	
(1)	{90} {88} {89} {92}
(2)	{98} {12} {23} {22}
(3)	{20} {22} {33} {0A}
(4)	{21} {19} {9A} {21}
Key_R2	71
Nipple_key	7

Op_code	DD72
CLK	1
Clockwise	FALSE
Matrix_E	
(1)	{21} {11} {13} {24}
(2)	{31} {12} {23} {44}
(3)	{41} {22} {33} {14}
(4)	{42} {32} {34} {43}

Op_code	DD82
CLK	1
Hi_Key	FALSE
Matrix_E	
(1)	{21} {11} {13} {43}
(2)	{31} {12} {23} {44}
(3)	{41} {22} {33} {14}
(4)	{42} {32} {34} {24}
Key_R2	71
Nipple_key	1

Op_code	DD61
CLK	1
Clockwise	FALSE
Matrix_E	
(1)	{11} {12} {43} {44}
(2)	{21} {22} {13} {14}
(3)	{31} {32} {23} {24}
(4)	{41} {42} {33} {34}

Op_code	DD91
CLK	1
Clockwise	FALSE
Hi_Key	FALSE
Matrix_E	
(1)	{11} {12} {13} {14}
(2)	{21} {22} {23} {24}
(3)	{31} {32} {33} {34}
(4)	{41} {42} {43} {44}
Key_R1	E3
Nipple_key	3

Figure (19) snap shots of the decryption steps

6. Conclusion

In this paper, a VHDL module was introduced to carry out various light block-cipher protocols rapidly for 128 bits within 4X4 matrix.

The behavior of this module has been verified through a full short scenario. Any cryptography programmers can extend our mentioned scenario up to thousands of VIs (or more) or reduce it according to their design requirements and degree of security.

The speed of performing these proposed cipher protocols is determine by one VI/clock pulse.

Later, designers and researchers can modify this idea with various features such as: -

- Expanding its matrix dimensions to handle a wide range of data simultaneously.
- Adding more VIs with different ideas of cipher protocols.
- Assigning a greater number of key-bytes.
- Implementing it by the VLSI tools to consume little power.

References

[1] M. Saleh, N.Z. Jhanjhi, A. Abdullah and R. Saher, "Design Challenges of Securing IoT Devices: A survey", International Journal of Engineering Research and Technology, Vol. 13, No. 12, 2020, pp. 5149-5165. http://www.irphouse.com/ijert20/ijertv13n12_149.pdf

[2] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin and C. Vikkelsoe, "PRESENT: An Ultra-lightweight Block Cipher", Cryptographic hardware and embedded systems-CHES 2007, Springer, Vol. 4727, 2007, pp. 450–466. https://doi.org/10.1007/978-3-540-74735-2_31

[3] R. Beaulieu, S.T. Clark, D. Shors, B. Weeks, J. Smith and L. Wingers, "The SIMON and SPECK lightweight block ciphers", 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, Number:15311795, June 2015, pp. 1–6. https://doi.org/10.1145/2744769.2747946

[4] "The 128-bit Blockcipher CLEFIA: Algorithm Specification", Sony Corporation, 2007, pp.1-41. Retrieved from https://www.sony.co.jp/Products/cryptography/clefia/download/data/clefia-spec-1.0.pdf

[5] J. Daemen and V. Rijmen, "The Advanced Encryption Standard Process", The design of Rijndael, Springer, 2002, pp.1-8. https://doi.org/10.1007/978-3-662-04722-4_1

[6] D. Kwon, J. Kim, S. Park, S.H. Sung, Y. Sohn, J.H. Song, Y. Yeom, E-J. Yoon, S. Lee, J. Lee, S. Chee, D. Han and J. Hong, 'New block cipher: ARIA', Information Security and Cryptology - ICISC 2003, Springer, Vol. 2971, 2004, pp. 432–445. https://doi.org/10.1007/978-3-540-24691-6_32

[7] P. Kitsos, N. Sklavos, M. Parousi and A. Skodras, "A comparative study of hardware architectures for lightweight block ciphers", Computers and Electrical Engineering, Vol. 38(1), 2012, pp. 148-160. https://doi.org/10.1016/j.compeleceng.2011.11.022

[8] N. sklavos, A. priftis, P. kitsos and O. koufopavlou, "Reconfigurable crypto-processor design of encryption algorithms operation modes: methods and FPGA integration", 2003 46th Midwest Symposium on Circuits and Systems, IEEE, Number:8814189, 2003, pp. 811-814. https://doi.org/10.1109/mwscas.2003.1562410

[9] F. Pirpilidis, L. Pyrgas and P. Kitsos, "8-bit Serialised Architecture of SEED Block Cipher for Constrained Devices", IET Circuits, Devices and Systems, Vol. 14 (3), 2020, pp. 316-321. https://doi.org/10.1049/iet-cds.2018.5354

[10] Assem Badr, "Awesome back-propagation machine learning paradigm", Neural Computing and Applications, Vol 33, 2021, pp. 13225-13249. https://doi.org/10.1007/s00521-021-05951-6

[11] A. Badr, A.M. Fouda and A. kodb, "Modify the µCS-51 Architecture to SIMD, VLIW and Superscalar µC", International Journal of Computer Science Issues, Vol 9 (1), 2012, pp. 121-128. https://www.ijcsi.org/papers/IJCSI-9-1-1-121-128.pdf

[12] A. Badr, A.M. Fouda and A. kodb, "Modify the µCS-51 with Vector Instructions", International Journal of Computer Science Issues, Vol. 9 (3), 2012, pp.165-174. http://www.ijcsi.org/papers/IJCSI-9-3-3-165-174.pdf

[13] A.M. Fouda and A.Badr, "Design modified architecture for MCS-51 with innovated instructions based on VHDL", Ain Shams Engineering Journal, Vol. 4(4), 2013, pp.723-733. https://doi.org/10.1016/j.asej.2012.12.001

[14] E. Roy, 2021, AVR Memories [Video]. https://microchipdeveloper.com/8avr:memory

[15] "8-bit Atmel Microcontroller with 128Kbytes In-System Programmable Flash", Atmel Corporation, 2007, pp.1-141. Retrieved from http://ww1.microchip.com/downloads/en/devicedoc/doc0945.pdf