PROJECT REPORT
ON

FPGA IMPLEMENTION OF AES AND CHACHA20 CRYPTOGRAPHIC ALGORITHM

Submitted To:
Dr. Gaurav Trivedi

Mentored By:
Miss. Meenali Janveja

Submitted By:
Kautilya Joshi (IIIT NAGPUR)
Utkarsh Bhiogade (IIIT NAGPUR)

ACKNOWLEDGEMENT

We express our deep sense of gratitude to professor, ***Dr.Gaurav Trivedi***, and our Mentor ***Miss.Meenali Janveja*** for there valuable guidance and encouragement. We would also like to express our gratitude to all those who were involved directly or indirectly with the completion of this project.

Contents

1 AES Algorithm

1.1 About The Paper

In this paper[1], using Advanced Encryption Standard (AES) Algorithm, the author has proposed an optimized key expansion module for secure transmission of ECG signals. AES is a symmetric, non-fiestal block cipher cryptographic algorithm that encrypts and decrypts the data block on 128 bits using different key sizes. Based on the block sizes, the number of rounds of encryption and decryption operations and the number of subkeys generated from the main key differs. In the proposed algorithm the subkey generation is altered to speed up the process of generating subkeys from the main key. To decrease the time consumption for obtaining the subkeys required to convert the plain text into non-intelligent cipher text, a pipelined architecture was implemented. By this proposed algorithm, the author found that there was 50% reduction in the time taken to generate all the subkeys.

1.2 AES Algorithm

AES serves as a most important cryptographic algorithm which satisfies the most important security goals (Confidentiality, Integrity, Availability) for the secure communication over an unsecure communication channel. The key lengths which are used for encryption and decryption of data in AES is associated with the number of rounds used in the AES architecture. The size of data block is same for all version of AES algorithm. Each different subkey which is generated from the single main key is used in each round of encryption and decryption.

1.2.1 AES Encryption and Decryption

The steps involved in the algorithm of Encryption and Decryption are as follows:

1. Sub-byte/Inverse Sub-byte
   In this step, each byte of plain text array is replaced with a SubByte using 8-bit Substitution box.It is replaced in hexadecimal byte using substitution table/Inv Subtution Table for Encrytion/Decryption Process respectively. It is the only Non-linear process in the algorithm.

2. The Shift Row Step
   In this step the rows in the state arrays are shifted left cyclically by a certain offset. The first row remains unchanged as the offset in zero. The second, third and fourth row are shifted by offset of 1,2 and 3 respectively. The columns of output which is obtained after this step is composed of bytes from each column from the input state.In case of Decryption, the rows are shifted cyclically right with respective offset value.The importance of this step is to avoid the columns being encrypted independently, in which case AES degenerates into four independent block ciphers.

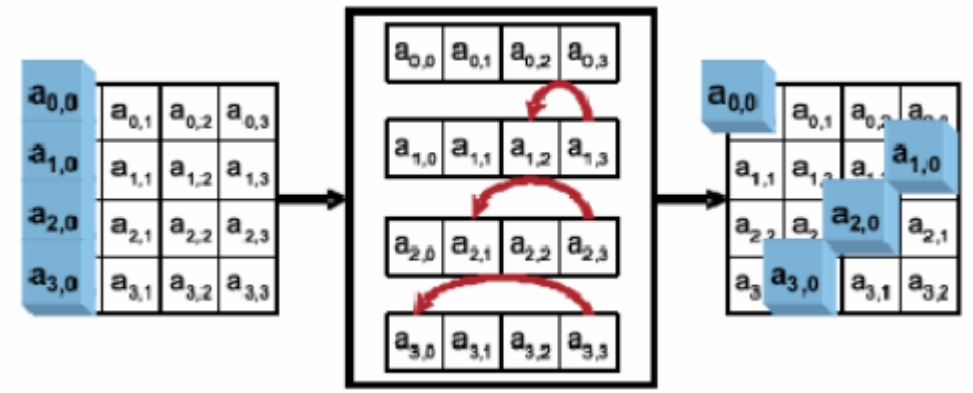


Figure 1: Shift Row

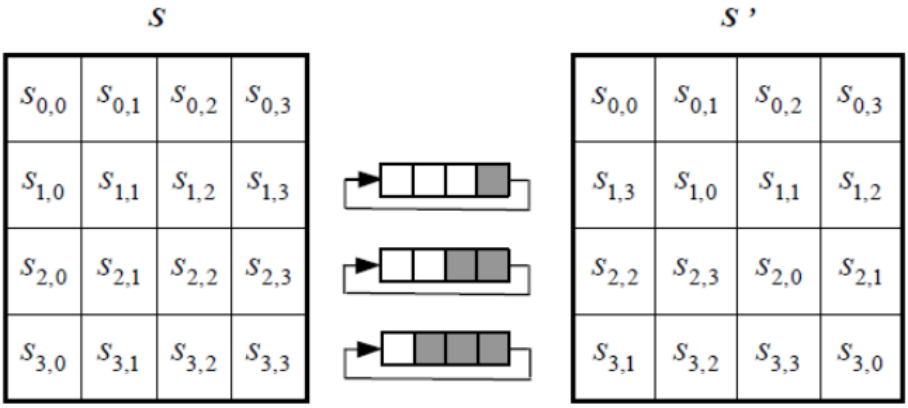


Figure 2: Inverse Shift Row

3. Mix columns/ Inv Mix columns
   In this Step, column level operations take place, a simple way to obtain is Matrix Multiplication
   For Mix Column operation

$$\begin{bmatrix} S'1 \\ S'2 \\ S'3 \\ S'4 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S1 \\ S2 \\ S3 \\ S4 \end{bmatrix}$$

   For Inv Mix Column operation

$$\begin{bmatrix} S'1 \\ S'2 \\ S'3 \\ S'4 \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} S1 \\ S2 \\ S3 \\ S4 \end{bmatrix}$$

   Matrix multiplication is composed of multipliaction and addition of entries. Here adding is simply XOR

4. Add Round Key
   In this step,the subkey is combined with each state. Then this subkey (Round Key) is simply added to the State array by a bitwise XOR operation.

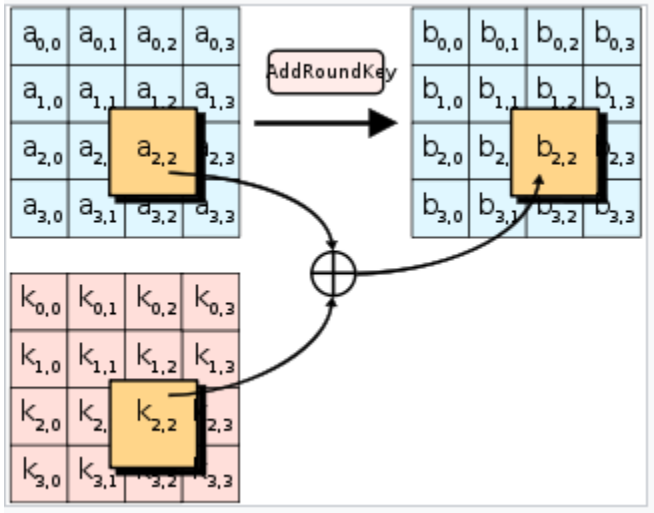


Figure 3: Add Round Key

1.2.2 Existing Architecture for Key-Expansion

In this process round key are generated which are used in each round.If the number of rounds is N_r, then the Key-Expansion routine creates $N_r + 1$ round keys (128 bit) from single cipher key (128 bit). It creates round keys word by word, where a word is an array of four bytes. In the Existing Architecture $4(N_r + 1)$ words are obtained from the intial main key, W_0,W_1,W_2,W_3. From Fig. 4, it is clear that each word created purely depends on the word at the left and the word at the top if it i $\neq 4$.
$If i = 4$ditemize , then the current word also depends on a temporary word, which is the result of subword and rotwr- n $W_{i1}$and XORing the result with R con. Where N_r = number of rounds and i = word number in key. Temporary word = (Rotword(W_{i1})) Rcon

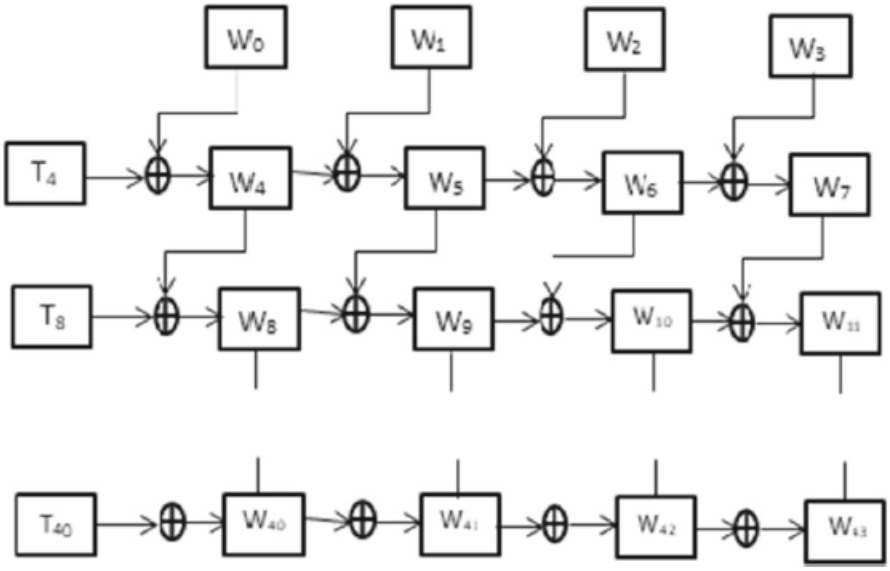


Figure 4: Existing Architecture

The table below shows the R constanst (Rcon) values for diffrent rounds in AES-128

Round	Rcon	Round	Rcon
1	$(01000000)_{16}$	6	$(20000000)_{16}$
2	$(02000000)_{16}$	7	$(40000000)_{16}$
3	$(04000000)_{16}$	8	$(80000000)_{16}$
4	$(08000000)_{16}$	9	$(1B000000)_{16}$
5	$(10000000)_{16}$	10	$(36000000)_{16}$

Table 1: Rcon values for diffrent rounds in AES-128

1.2.3 Proposed architecture

In the existing architecture, each word in the current subkey depends on the previous subkey. Therefore all the subkeys are generated in sequential manner, one after the other. Generation of last subkey takes place after the completion of the generation of all previous subkeys.This process consumes lots of time to generate all the subkeys from main key, since all the subkeys cannot be determined simultaneously. This is the major drawback of the existing architecture of Key expansion in AES algorithm.

In the proposed architecture, the time consumption was significantly reduced. Here, the whole architecture was splitted into two block which were executed in parallel. In the new architecture too, the process of generation of temporary words remains same as that of existing arrchitecture.

In this architecture, sixth subkey is generated from the main key instead of fifth key, and because of this modification, sixth key will be generated at the same time when first subkey is generated. Hence, both blocks can generate subkeys in parallel.

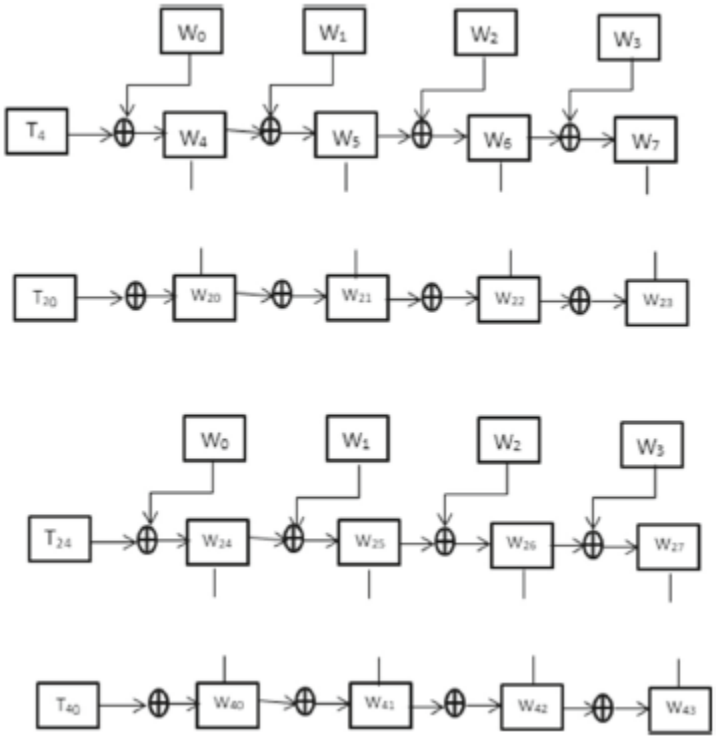


Figure 5: Proposed Architecture

It is clear that in the proposed architecture t_{24} is generated from W_2, where as in existing architecture t_{24} is generated from W_{23}. Also sixth subkey is fully dependent on the main key instead of fifth subkey.

1.3 Code explaination

CODE EXPLANATION:
A) ENCRYPTION
• An sbox module is created which takes the input as an 8 bit value where the upper nibble is the row index and lower nibble is column index. The output of this module is the value to be substituted in the subbyte transformation.
• In Subbytes module each byte of the plain text given is substituted using sbox.
• A Shiftrow module is created to perform the shift row operation as described above.
• A mixcolumn module is created to perform the mix column function as described above. A function is defined in this module in which each bit of the output can be expressed in terms of the input bits.
• A KeyGen module is created to perform the above described key generation process.

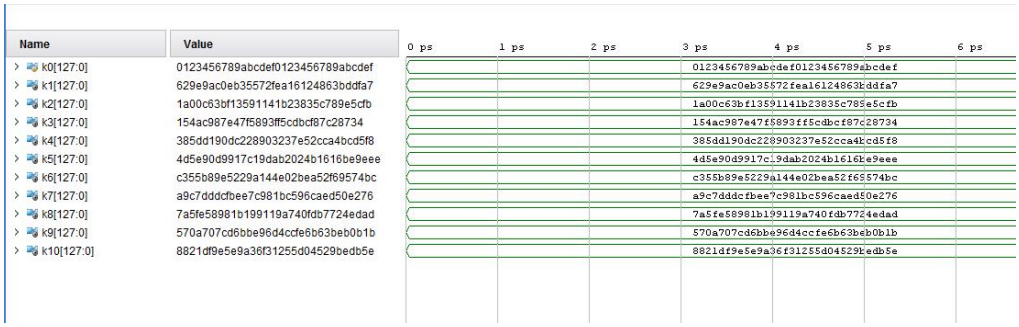
• A rounds module is created in which the following operations are done sequentially:
I) Subbytes transformation to the input data by using SubBytes module.
II) The rows are shifted using ShiftRow module.
III) Mixcolumn operation is done based on mixcolumn module.
IV) The output of rounds module is obtained by XORing the output of (III) with the respective key input.

• A roundlast module is created in which the following operations are done sequentially:
I) Subbyte transformation to the input data by using Subbyte module.
II) The rows are shifted using Shiftrow module.
III) The output of rounds module is obtained by XORing the output of (III) with the respective key input.
• A cipher module is created which performs the following
a) All the subkeys are produced using KeyGen module.
b) The rounds module is used in which the first round has a input which is obtained by XORing the input with the secret key. For the rest of the rounds the input will be outputs of the previous rounds.
c) For the last round roundlast module is used and output of this module is the output of cipher module.

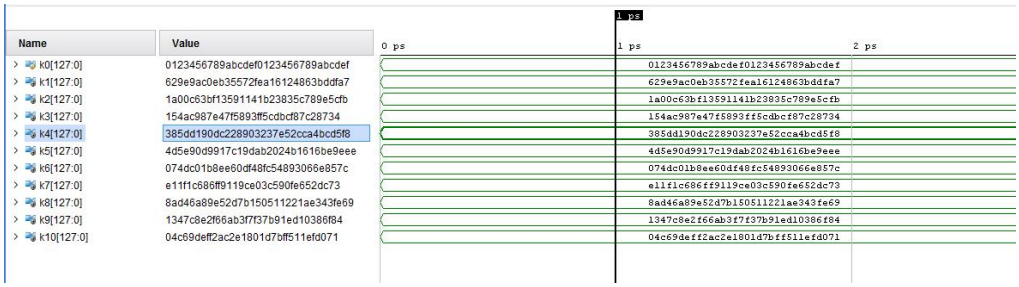
The input image file is converted into a text file (input.txt) using a MATLAB code .
• A read module is created to read input.txt file and store it in an array.
• A aestop module is created which reads the given input file data by the read module and the output of this module is given as the input to the cipher module.
• A Test bench is created (tb_encryption) for this top module where the secret key and the input data are given and the produced output is stored in a text file (encrypted.txt).

B) MODIFIED AES:
• In the modified AES algorithm , the image encryption and decryption is done in the same way of image encryption and decryption algorithm of standard AES. Here only the key generation process varies where round 6 uses secret key as the parent key.

1.4 Result



(a) Key Generation with Existing Architecture



(b) Key Generation with Modified Architecture

Figure 6: Simulation Results of Key Generation

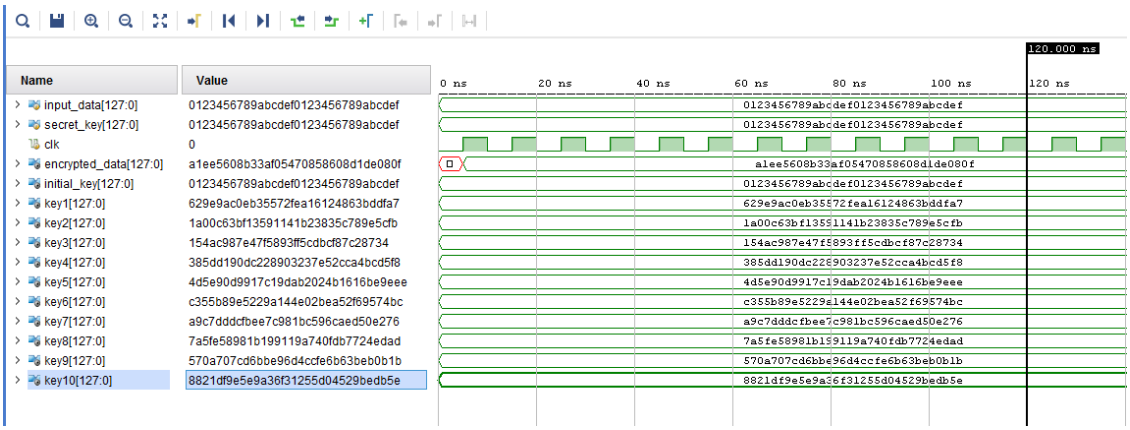


Figure 7: Simulation Results of Encryption

1.5 References

1. FPGA implementation of an optimized key expansion module of AES algorithm for secure transmission of personal ECG signals. *Thanikodi Manoj Kumar, Palanivel Karthigaikumar*

2. `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

3. Cryptography and Network Security by Behrouz A. Forouzan

4. `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`

2 CHACHA20 Algorithm

Implementation of CHACHA20 cipher for Encryption and Decryption of Data as proposed by Daniel J. Bernstein in "ChaCha, a variant of Salsa20".

2.1 CHACHA20 Algorithm

2.1.1 Introduction

This is a high-speed cipher first described by Daniel J. Bernstein in "ChaCha, a variant of Salsa20"[1].Chacha20 is based on Salsa20's 20-round cipher with changes to design algorithm that improve diffusion per round, consequently improving time requirement for each round and increasing resistance to attackers.

2.1.2 Elaboration

Algorithm[2] is explained from bottom to up, firstly 3 basic operation on 4-byte words, which builds up Quarterround, continuing through formation and updation of State-matrix, and finishing with encryption of plaintext input data.

Words

In the Algorithm **word** is defined as an element if $0, 1, ..., 2^{32} - 1$, they are written in hexadecimal and denoted by $0x$: For example, $0xabcdef98 = 11.2^{28} + 12.2^{24} + 13.2^{20} + 14.2^{16} + 15.2^{12} + 16.2^8 + 9.2^4 + 8.2^0 = 2882400152$ 3 Basic Operations:

- **Sum** of two word a, b denoted by $a + b = (a + b) \bmod 2^{32}$. For example, $0xc0a8787e + 0x9fd1161d = 0x60798e9b$.
- **Exclusive-or** of two words a, b denoted by $u \oplus v$, is sum of two number with carries suppressed. For example, $0xc0a8787e \oplus 0x9fd1161d = 0x5f796e63$
- **c-bit left rotaion** of a word a is denoted by $a <<< c = \sum_i 2^{i+c \bmod 32} a_i$. For Example, $0xc0a8787e <<< 5 = 0x150f0fd8$.

QuarterRound

It is the basic operation of the Chacha algorithm, it uses four 32-bit unsigned integers , denoted by a, b, c, and d. It is as follows (in C notation):

1. $a+ = b$; $d \oplus = a$; $d <<<= 16$;
2. $c+ = d$; $b \oplus = c$; $b <<<= 12$;
3. $a+ = b$; $d \oplus = a$; $d <<<= 8$;
4. $c+ = d$; $b \oplus = c$; $b <<<= 7$;

For example, Below showing 4 sample words their add, XOR, and rotate operation for fourth line.

- $a = 0x11111111$
- $b = 0x01020304$
- $c = 0x77777777$
- $d = 0x01234567$
- $c = c + d = 0x77777777 + 0x01234567 = 0x789abcde$
- $b = b \oplus c = 0x01020304 \oplus 0x789abcde = 0x7998bfda$
- $b = b <<< 7 = 0x7998bfda <<< 7 = 0xcc5fed3c$

Finally output after running whole QuarterRound.

- $a = 0xea2a92f4$
- $b = 0xcb1cf8ce$
- $c = 0x4581472e$
- $d = 0x01234567$

Chacha State

The algorithm consists of Chacha state matrix which is a 4×4 Matrix which keeps updating through QuarterRound, and finally, it is used for encryption of plain-text.
Initial State matrix is formed using following:

1. **Constant word** (denoted by **c**): First row of state matrix contains four constant 32-bit words specified in the algorithm which are:
   - $0x61707865$
   - $0x3320646e$
   - $0x79622d32$
   - $0x6b206574$
2. **Key** (denoted by **k**): Second and Third Row of state matrix is filled from 256-bit key which is treated as a concatenation of eight 32-bit little-endian order integer..
3. **Block Counter**(denoted by **b**): Fourth Row's First Column contains 32-bit counter initialized from 1 as little-endian integer.
4. **Nonce**(denoted by **n**): Fourth Row's last four columns contains 96-bit nonce which is treated as a concatenation of three 32-bit little-endian integer. Nonce is a random 96-bit data which should not be repeated for same set of key.

The Chacha State matrix:

ccccccc	*ccccccc*	*ccccccc*	*ccccccc*
kkkkkkkk	*kkkkkkkk*	*kkkkkkkk*	*kkkkkkkk*
kkkkkkkk	*kkkkkkkk*	*kkkkkkkk*	*kkkkkkkk*
bbbbbbbb	*nnnnnnnn*	*nnnnnnnn*	*nnnnnnnn*

Chacha State Matrix is updated in two manner depending on sequence of input from state matrix in QuarterRound, that are:
Column rounds:

1. QUARTERROUND (0, 4, 8,12)
2. QUARTERROUND (1, 5, 9,13)
3. QUARTERROUND (2, 6,10,14)
4. QUARTERROUND (3, 7,11,15)

Diagonal rounds:

1. QUARTERROUND (0, 5,10,15)
2. QUARTERROUND (1, 6,11,12)
3. QUARTERROUND (2, 7, 8,13)
4. QUARTERROUND (3, 4, 9,14)

2.1.3 Steps of Chacha20 Algorithm

1. Initial State Matrix is created using constant, keys, block count, and nonce.
2. Initial State Matrix is passed as Column Rounds in QuarterRound updating the State Matrix with output, which is then again passed as Diagonal Rounds in QuarterRound updating the State Matrix.
3. Like this Column Round and Diagonal Round are performed alternatively for 10 cycles each.
4. Logically OR operation is performed between Final State Matrix and Initial State Matrix.
5. Output State Matrix is serialized in little-endian order, in 4-byte chunks.
6. In the Final Step, two chunks of 32-bits are taken from State Matrix and plain-text data input is taken in sequence of 64-bits and XOR operation is performed.
7. Final output sequence is the encrypted text

NOTE: For Decryption, input Data is replaced with the Encrypted text, without changing any other function, output obtained would be Decrypted text.

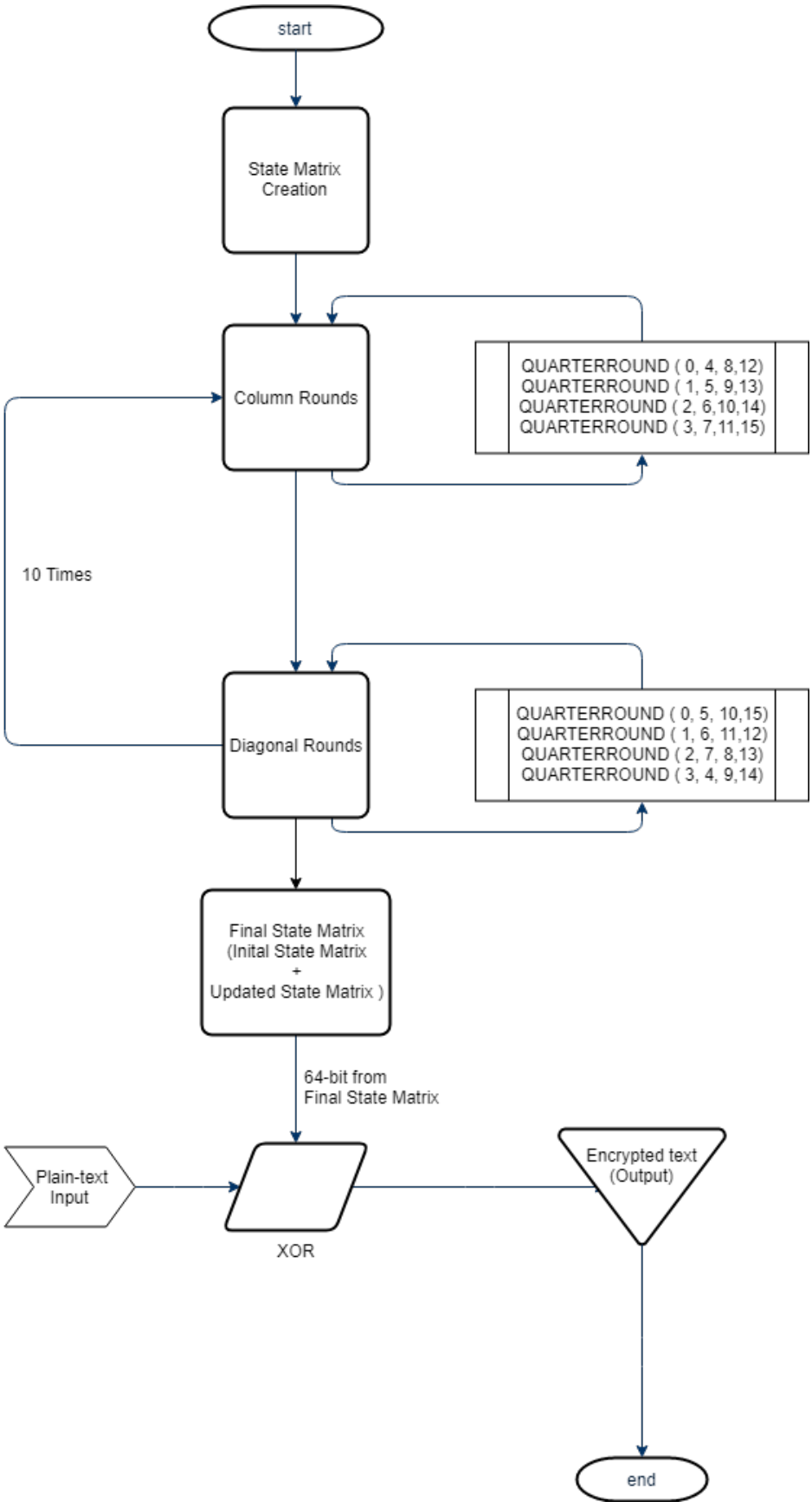


Figure 8: Flowchart of CHACHA20 Algorithm

2.2 Code Explanation

1. There are three main modules in the CHACHA20 project and a Testbench which drive the code. The whole functionality of the code is controlled by a FSM (Finite State Machine).

2. The basic module is the "chacha_qr.v" module in which the Quarterround Operations are performed as described in algorithm i.e, a Column round and a Diagonal round which consits of a 32-bit addition, 32-XOR operation and Cyclic-left shift operation by 16, 12, 8 and 7 bits respectively.

3. The module "chacha_core.v" drives the FSM. Various Control States are defined in this module which represent different States of the FSM. These states are then controlled by various flags.

4. The constants used to form the state matrix can be changed as given in the algorithm. A function named "12b" is defined to convert the 32-bit Hex pattern from big endian to little endian.

5. In first always block, the flag states are being modified based on the other control flags. In 2nd always block, the initial state matrix is formed. In 3rd always block, the values to be passed in the quarterround module are initialized with zero and then based on various control flag status, the values are passsed in "chacha_qr.v" module and also are copied to a temporary state matrix.

6. Now the states which are obtained are in Big Endian form and need to be converted to little endian after adding them with the initial state matrix and this is done in another always block. After adding them and converting to little endian, this state is then XORed with input data to be encrypted. In another always block the quarterround counter is incremented based on various flags which drives the methodology of quarterround, i.e. first column round and then diagonal round. There are two other always blocks which increments the quarterround counter and round counter respectively.

7. The top module driving "chacha_core.v" is "chacha.v" module, i.e., "chacha_core.v" is instantiated here. In this module, various local parameters have been declared to represent the states, various registers to represent the flags used and also various registers and wires which are required in the intermediate states.

8. The inputs passed in "chacha_core.v" module are controlled by two always blocks. 1st always block runs on every positive edge of clock. Main function of this block is to update the various registers and also update the value of flags. The 2nd block runs considering all the variables in that block to be in sensitivity list i.e., it'll function as a combinational logic.

9. Based on the status of various control flags and addresses which are passed, the corresponding flag values are modified which in turn modifies the value in the registers and then these values of registers are passed in the "chacha_core.v" module.

10. The whole code is then run on specific inputs given in testbench. There's a task named "run_test_vector" which is been called in the testbench. The 256-bit Key, 96-bit Nonce, Number of Rounds to be performed and 512-bit Input data is provided there. Also there are various functions/tasks written to display the data, and for various functionalities which are required in the code.

11. Various local parameters are defined to represent the various states used in the algorithm and also the states of the State Matrix are mapped to these parameters (consider them as indexes of the State Matrix).

12. Now the database is loaded in MATLAB software. This data is then exported to a text file in required format.

13. The input for the testbench is taken from the text file obtained above Firstly, the data from the input file is imported in an array and then this is modified in such a way that a new array is formed whose each element is having the size of 512-bits. This is then given as input to the task using a for loop.

14. The output values returned after execution of the code are displayed and stored in a text file.

15. This file is then again imported in MATLAB and the signal is plotted to see the results.

NOTE: The same code can be used for Encryption as well as decryption. When we provide plain text input data, we get encrypted data as output and when we provide the encrypted data as input, decrypted data is obtained as output.

2.3 Modifications

Few modification were done in the size of the State Matrix block which were XORed with the input data. In the original algorithm, the author used 64-bit of the key (obtained from the State Matrix) for XORing with the input data. In the modified code, the XORing was done with 32-bits, 64-bits, 128-bits, 256-bits and 512-bits of the State Matrix key. This modification made a significant change in the number of components required during implementation. Also a change was made which made it possible to give input greater than 512 bits.

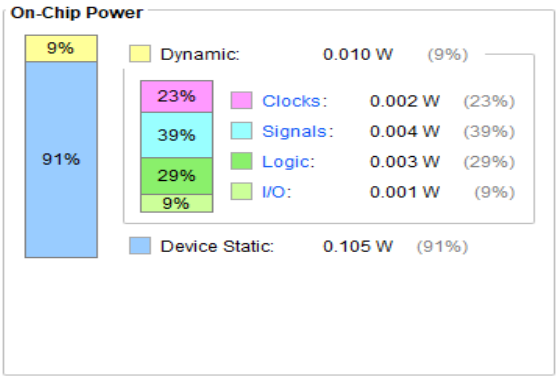
2.4 Results

To obtain the results, Xilinx Vivado® Design Suite Evaluation and WebPACK 2018.3 and MATLAB® R2018a were used.
The encryption of the Dataset was carried out using 32-bit, 64-bit, 128-bit, 256-bit and 512-bit XOR operation. During Synthesis of the above code for 32-bit, 64-bit, 128-bit, 256-bit and 512-bit XOR operation, significant changes in the utilization of components was observed. The utilization of different components and the Worst Negative Slack (WNS) values of respective architectures is listed in table 2.

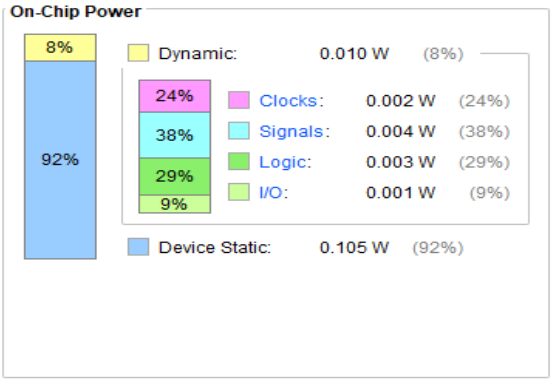
S. NO.	Architecture	Slice LUT Utilization	WNS
1	$32-bit$	2064	$0.601ns$
2	$64-bit$	2078	$1.081ns$
3	$128-bit$	2142	$1.018ns$
4	$256-bit$	2270	$0.944ns$
5	$512-bit$	2745	$1.051ns$

Table 2: Resource Utilization and WNS values for different Architectures

It can be clearly seen that the architecture in which 32-bit XOR operation was carried out has very less utilization of Slice LUTs as compared to other architectures. This less utilization of resources then reduces the area of the chip. Hence it can be found that the architecture with 32-bit XOR operation will consume less area as compared to the other architecture.
Based on the architecture, the chip will also have power requirement. The Power requirement report as obtained after implementation of the code is shown in fig 8. It was found that 32-bit, 256-bit and 512-bit architectures had same power requirement and similarly, 64-bit and 128-bit architecture had same power requirement. But the 32-bit, 256-bit and 512-bit architecture had more dynamic power as compared to 64-bit and 128-bit architecture. Though the difference between the Dynamic power of the two reports was very less, it is recommended to use the architecture with more dynamic power.



(a) Power requirement for 32-bit, 256-bit and 512-bit architecture



(b) Power requirement for 64-bit and 128-bit architecture

Figure 9: Power Requirement

Hence, it is recommended for the user to use 32-bit architecture as it has more dynamic power and also it has less Utilization of resources (LUTs) as compared to other architectures.

2.5 References

1. ChaCha20 and Poly1305 for IETF Protocols by Yoav Ni
2. ChaCha, a variant of Salsa20 by Daniel J. Bernstein
3. Salsa20 specification by Daniel J. Bernstein
4. Refrence Code For Chacha20 by Joachim Strömbergson