Documentation of AEAD and Compression Algorithms - A Verilog Implementation

submitted in partial fulfillment of the requirements

for the degree of

BACHELOR OF TECHNOLOGY

in

ELECTRICAL ENGINEERING

by

AMBATI SATHVIK	**EE17B002**
TIRUNAGARI RAHUL	**EE17B030**

Supervisor(s)

Dr. Vikramkumar Pudi

भारतीय प्रौद्योगिकी संस्थान तिरुपति

TIRUPATI

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY TIRUPATI

JUNE 2021

DECLARATION

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission to the best of our knowledge. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Place: Tirupati	**Signature**
Date: 02-05-2021	A.Sathvik
	EE17B002

Place: Tirupati	**Signature**
Date: 02-05-2021	Tirunagari Rahul
	EE17B030

BONA FIDE CERTIFICATE

This is to certify that the report titled **Documentation of AEAD and Compression Algorithms - A Verilog Implementation**, submitted by **Ambati Sathvik & Tirunagari Rahul**, to the Indian Institute of Technology, Tirupati, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the project work done by them under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Tirupati
Date: 02-06-2021

Dr. Vikramkumar Pudi
Guide
Assistant Professor
Department of Electrical Engineering
IIT Tirupati - 517501

Note: If supervised by more than one professor, professor's name must be included and get signatures from all supervisors. Change him/her or our or my accordingly.

ACKNOWLEDGMENTS

We would like to express our gratitude to Dr. Vikramkumar Pudi for giving his valuable suggestions during the planning and development of this work. His guidance was extremely helpful and constructive.

ABSTRACT

KEYWORDS: Cipher; Authentication; Compression; Post-quantum cryptography.

The applications of security are addressed through cryptography primitives and they find vast applications from web sites to safe guarding nation's classified data. Many of the attack resistant algorithms demand considerable area and also consume adequate power. Many of the modern day requirements seek secured strategies, that can be executed using limited power and within small time span & area. In this regard, light weight ciphers, compression techniques are understood and simulated.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **i**

ABSTRACT **ii**

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ABBREVIATIONS **vii**

NOTATION **viii**

1 INTRODUCTION **1**

2 Literature Review **2**

3 Types of cryptography **4**

3.1 ACORN 5

3.2 Sponge Capacity (SpoC) 6

3.3 WAGE 8

4 Hardware implementation **11**

5 Compression **13**

5.1 Overview 13

5.2 Preprocessing 14

5.3 Discrete Cosine Transform 15

5.3.1 1D DCT 15

5.3.2 2D DCT 16

5.3.3 Importance and significance 18

5.4 Quantisation 19

5.5 Zigzag and Run-length encoding 20

5.6 Huffman coding 21

6 Results and Discussions **25**

7 SUMMARY AND CONCLUSION **26**

A Colour space **29**

B Zigzag **30**

C Verilog implementation **31**

C.1 DCT 31

C.1.1 Determining M.T’ 32

C.1.2 Determining T.M.T’ 32

C.2 Quantisation 33

C.3 Run-length encoding 33

LIST OF FIGURES

3.1 ACORN state update 5
3.2 ACORN message and control strings 6
3.3 Step permutation 7
3.4 Simeck box(SB) and Sub Simeck box (Sb) 7
3.5 Processing associated data 8
3.6 Processing plaintext during encryption 8

4.1 Top level block diagram of LWC core 12

5.1 Bases of 2D-DCT 18

B.1 The zigzag sequence 30

LIST OF TABLES

3.1 ACORN component information 6
3.2 ACORN design information 6
3.3 WAGE component information 9
3.4 WAGE design information 9

5.1 Family table 22
5.2 Prefix-free *code_L* for ac terms 24

6.1 Cryptocore component information 25
6.2 Utilisation design information 25

ABBREVIATIONS

AD	Associated data, this optional data will not be encrypted or decrypted.
CT	Ciphertext is the encrypted message.
DCT	Discrete cosine transform.
MCU	Minimal coded unit is an 8x8 block of an image.
Npub	Data that is valid only for an instant (also referred as nonce, public message).
PT	Plaintext is the message before encryption.

NOTATION

$x \oplus y$	XOR of bitstrings x and y.
$x \parallel y$	Concatenation of bitstrings x and y.
$x << i$	Cyclic rotation of bitstring x to the left by i bits.
$x \odot y$	Bit wise AND of bitstrings x and y.
$adlen$	Bit length of the associated data.
$pclen$	Bit length of the plaintext / ciphertext.
0^r	Bitstring 0 of length r.
1^r	Bistring 1 of length r.

CHAPTER 1

INTRODUCTION

The necessity to transfer information securely over an insecure channel gave rise to encoding messages so that only the expected people have access to the sensitive information, and unauthorized people could not impart anything, even if they intercept these encrypted messages.

Cryptography is the art and science of concealing the true meaning of messages, thereby introducing secrecy. The meaning of 'cryptography' is 'hidden meaning'. It encounters problems such as authentication, key establishment. The classic cryptography manipulates letters and digits directly, and the knowledge of the implemented algorithm persists only among the legitimate users.

The modern cryptography operates information at the bit level, and these algorithms (involving various concepts of mathematics that include number theory, computational complexity theory, and probability theory) are open to public, but the secrecy remains in the key. The progress in the field welcomed applications ranging from social media apps to secure transactions, time-stamping to SIM card authentication.

However, in real-world problems due to technological advancements, in addition to security, integrity and authenticity, one must also be able to transmit the obtained information compactly. Especially when working with the data dealing with signals like audio & video, our eyes and ears cannot distinguish subtle changes. These unnoticeable details can be eliminated to save up storage tremendously. We use compression techniques to eliminate this unwanted data. This lets us have shorter plain text and cipher text. Hence, time gets reduced while encrypting, decrypting and transmitting the data.

CHAPTER 2

Literature Review

Mukhopadhyay (2014) explained the origin of cryptoscience, and covered various traditional and modern ciphers. It explained how network professionals use cryptography to maintain the privacy of the data. Singh (2019) elaborated the evolution of hieroglyphs to the Enigma, Caeser cipher to AES and Turner (2019) summarised the encoding techniques.

Goldwasser and Bellare, Kothari (2020) explained functioning of crypto-algorithms, their usage, the risks associated with using them, and why governments should be concerned. Piper and Murphy (2013), Stallings (2006), highlighted important areas such as Stream Ciphers, block ciphers, public key algorithms, digital signatures, and applications such as e-commerce.

Authentication is the process of verifying the identity of the sender. Authentication algorithms also verifies the integrity of the secrets. The same is detailed in Juniper (2020), A.R. (2016), Rijmen (2011), McGrew (2008).

The algorithms that we deal in detail, are explained by Wu (2016) (ACORN), Aagaard *et al.* (2019) (WAGE), AlTawy *et al.* (2019) (SpoC). The in-house implementation of Sponge Cipher is illustrated in Kaps *et al.* (2019).

Advent of quantum computers make cryptanalysis much simpler. The RSA algorithm is solely based on prime factorisation, which is not a challenging task for quantum computers. To make the modern day algorithms quantum-proof, the idea of post-quantum cryptography evolved. These techniques are explained by Ahuja (2020).

Cabeen and Gent (2008) gives out the theoretical explanation of the JPEG compression process with the main basis on the DCT section. The important find of the paper is that the conversion of DCT equations into matrices form which really helps while coding. But it fails to clearly explain a proof of the same. It also does a comparative study on qualities of quantization matrices. Even though it says JPEG can be applied to both colour and grayscale images, it only focuses on grayscale images.

The article T.81 (1993) deals in detail with digital compression and coding of continuous-tone still images. It is a combined work prepared by Consultative Committee for International Telephony and Telegraphy (CCITT) study group VIII and the Joint Photographic Experts Group (JPEG). On a whole, it specifies processes for converting source image data to compressed image data, and to reconstructed image data, with necessary guidelines for implementation. It gives a complete picture of coded representations for compressed image data.

Jehhal Liu gave a brief understanding about JPEG compression. Mainly, this helped in understanding Run length encoding scheme. It also explains the difference between DC and AC coefficients. Zero-Runs and End-of-Block Indicators are also discussed here.

D'Angelo explains in detail about theoretical background of DCT. It uses MATLAB to help in clear understanding of the same. It also gives good visualisation of 2D-DCT. The article gave a base to understand and implement Huffman coding. This also introduces the concept of prefix free codes such that it can be uniquely decodable. Rhea gives a simple explanation of how JPEG compression works with an example. It performed compression on one MCU block of a given grayscale image.

CHAPTER 3

Types of cryptography

Encryption is the process of encoding sensitive information into gibberish text such that only the persons with an intended key can decrypt it back. It is possible to decrypt the message without the key, but, for a well constructed encryption scheme requires significant resources and skills to break.

Cryptographic algorithms are classified into three types based on the type of keys required. **Symmetric key** cryptography requires a common shared key to perform both encryption and decryption. AES and 3DES are approved symmetric key algorithms. AES is estimated as the most popular file encryptor with encoding over 50% of global data, including HTTPS protocols, nation's classified secrets, VPN servers, Zoom app, and till date, no practical attack was registered against the algorithm.

Asymmetric key algorithms use different keys to encrypt and decrypt information. Every user has a public key and a private key (so it is also known as public key cryptography). The public key is open to all, but the private key remains only with its user.

The data enciphered using the public key can only be decrypted back with its private key and vice versa. The former case makes sure that data can be accessed only by the owner of the private key, and the later one confirms the transmitter. Though these keys are cryptographically related, the private key cannot be derived from its public key.

Encryption protocol TLS, WhatsApp messenger, uses asymmetric and symmetric key cryptographic algorithms combinedly to ensure the confidentiality, integrity from the first primitive and authentication, non-repudiation from the second.

The **hash function** does not use a separate key to proceed. It digests of length arbitrary data to produce a fixed-length output known as the hash value. It is expected that the original message cannot be derived from its hash value, so it is handy to store passwords, to communicate a pre-shared key, used as the tag for authentication.

Authenticated encryption (AE) schemes provide confidentiality, integrity and authenticity to data at a time. The traditional way to achieve such a construction is to combine cryptographic primitives encryption for confidentiality and authentication for integrity, authenticity.

3.1 ACORN

ACORN is a lightweight AE stream cipher Wu (2016). The cipher consists of linear feedback shift register of length 293 and aims to protect up to 2^{64} bits of associated data and up to 2^{64} bits of plaintext and to generate t bit authentication tag by using a 128-bit secret key and a 128-bit nonce.

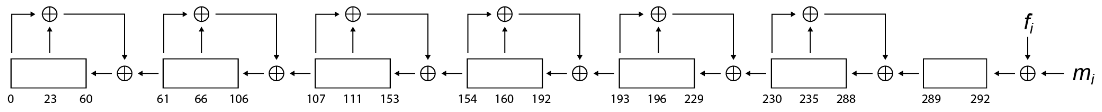

Figure 3.1: ACORN state update

Keystream bit ks_i and feedback bit f_i are generated from every state S_i, the obtained feedback bit f_i and the input message bit m_i update the state S_{i+1} (explained in fig.3.1), the new state S_{i+1} is function of it's previous state S_i, m_i, ca_i, and cb_i Wu (2016). The primary functions used to compute these are maj and ch.

$$maj(x;y;z) = (x\&y) \oplus (x\&z) \oplus (y\&z)$$

$$ch(x;y;z) = (x\&y) \oplus ((\sim x)\&z)$$

$$ks_i = S_{i,12} \oplus S_{i,154} \oplus maj(S_{i,235}, S_{i,61}, S_{i,193})$$

$$f_i = S_{i,0} \oplus (S_{i,107}) \oplus maj(S_{i,244}, S_{i,23}, S_{i,160}) \oplus ch(S_{i,230}, S_{i,111}, S_{i,66}) \oplus (ca_i\&S_{i,196}) \oplus (cb_i ks_i)$$

The message bits (m_i) and control bits ca_i, cb_i are obtained from the bitstrings m, ca, and cb respectively, shown in fig.3.2.The message string m include the key, nonce, AD, and PT. Control strings ca, cb differentiate between processing AD and PT. The data blocks 1 to 21 of fig.3.2 are of definite lengths except blocks $15^*(adlen)$ and $18^*(pclen)$. The first state S_{1792} is initialised with 0^{293}, then, state is updated $1792 + adlen + 256 + pclen + 256 + 768$ times (i.e., length of message string). The keystream bits of the last t updates make up the tag (128-bit tag is recommended).

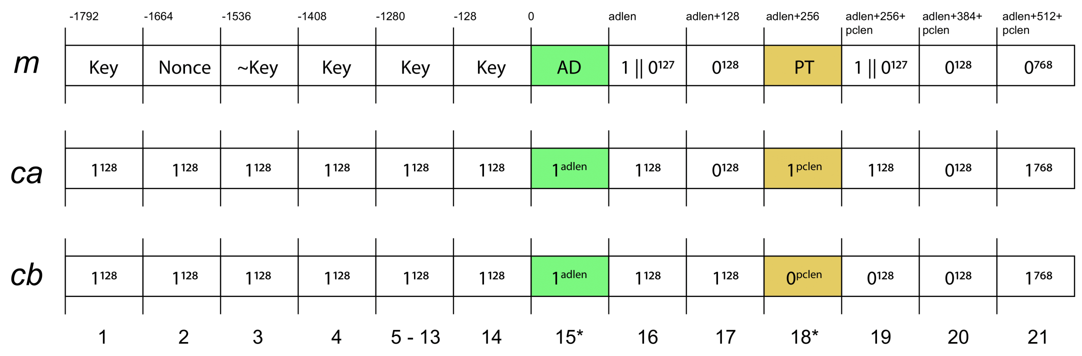

Figure 3.2: ACORN message and control strings

Specifications of algorithm with conventional 128-bit nonce, key AD, 200-bit message, are shown in tables 3.1, 3.2.

Component	size	number
Adders	2-input, 8 bit	2
	2-input, 9 bit	2
	2-input, 12 bit	3
XORs	2-input, 1-bit	74
Registers	1-bit	640
Muxes	2-input, 1-bit	1213

Table 3.1: ACORN component information

Site type	Used
IO pins	912
Flip Flops	653
LUTs	748
IO Buffers	912

Table 3.2: ACORN design information

3.2 Sponge Capacity (SpoC)

Sponge Capacity is a lightweight sponge-based cipher. SpoC-64 and SpoC-128 are the two variants of this algorithm. 64 and 128 are the rates at which information is fed and processed, both of them require 128 bit Key and Nonce. SpoC-128 is carried out with 256-bit state and generates 128-bit tag.

sLiSCP-light (step) permutation diffuses the information across the state as shown in fig.3.3, clear explanation of Simeck Box (SB) is illustrated in fig.3.4. The constants rc^a, rc^b, sc^a, sc^b are provided in AlTawy *et al.* (2019). State is represented as $S_0||S_1||S_2||S_3$, where S_i, 0 i 3, is 64-bit sub-state.

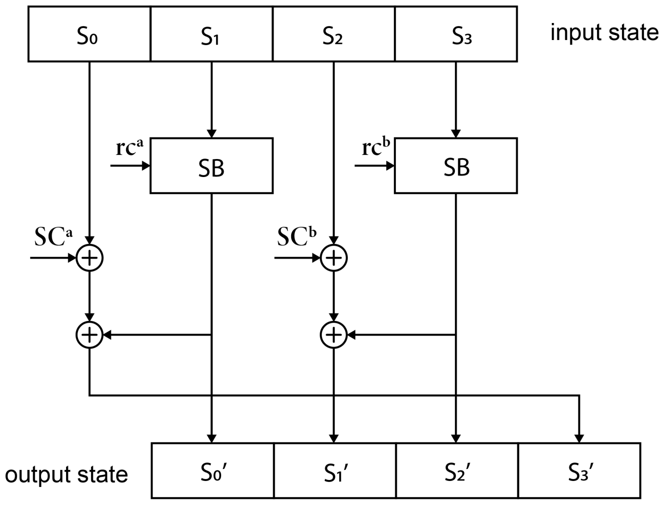

Figure 3.3: Step permutation

The state is initialised from 128-bit key $K_0||K_1$ and nonce $N_0||N_1$, as $N_0||K_0||N_1||K_1$. As 128-bit data can only be introduced into the state, associated data and plaintext are split into 128-bit blocks and the final block is padded (if necessary).

Control bits $(c_3||c_2||c_1||c_0)$ differentiate the complete and partial blocks, AD, plaintext and tag operations. The most significant bit c_3 is 1 during tag generation only. Bits c_2 and c_1 correspond to plaintext (also referred as Message block) and associated data and are 1 during their processes respectively.

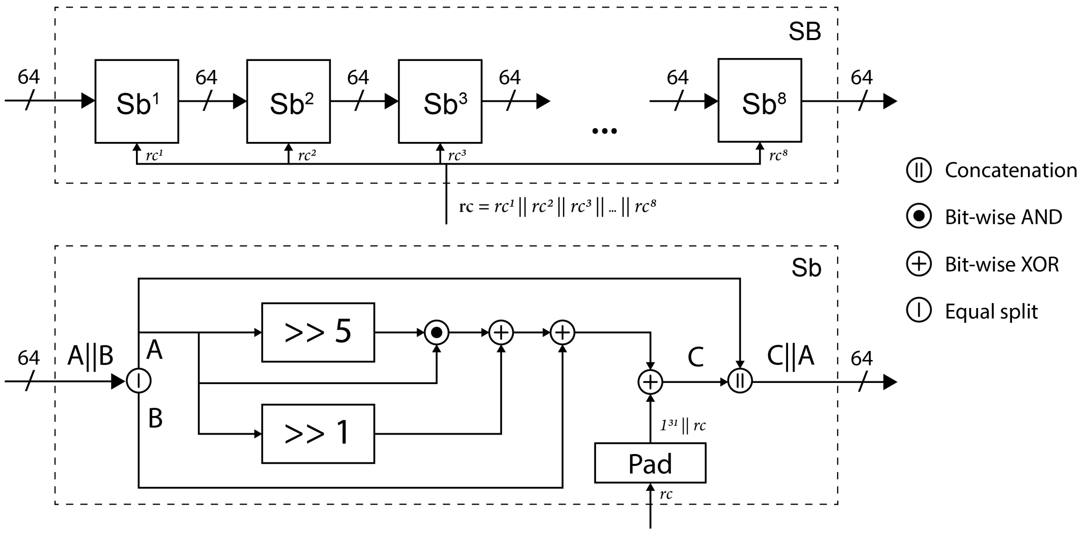

Figure 3.4: Simeck box(SB) and Sub Simeck box (Sb)

Processing associated data of length *adlen* starts with splitting AD into blocks of

128-bits (starting from MSB). If required, the last block is padded with $1||0^p$, $p = modulo(adlen, 128) - 1$ and the further process is described in fig.3.5.

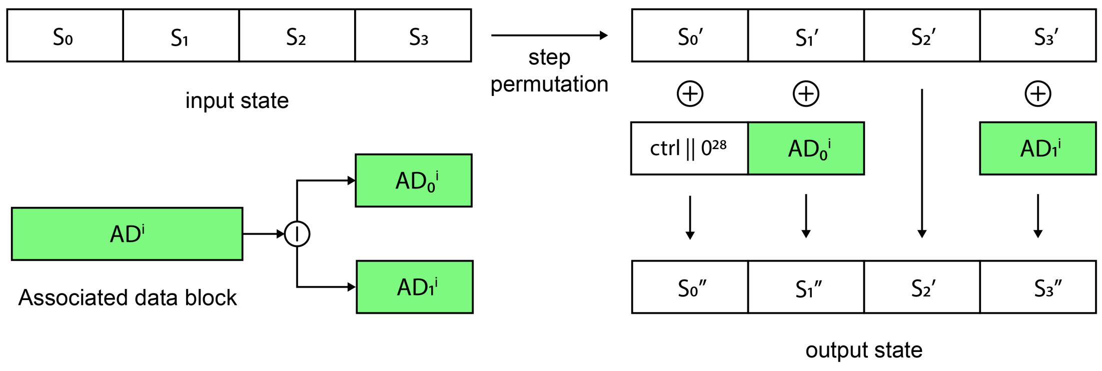

Figure 3.5: Processing associated data

Plaintext during encryption and ciphertext during decryption of length $pclen$, is padded (if required) and split into 128-bit blocks. During encryption, the input message block M^i is further split into 64-bit halves M_0^i, M_1^i and XORed with appropriate sub states $S_1^{'}$ and $S_2^{'}$. During the decryption process, the input ciphertext block CT^i is XORed with sub states $S_0^i||S_2^i$ and the obtained plaintext block M^i is split and the successive steps are same as encryption.

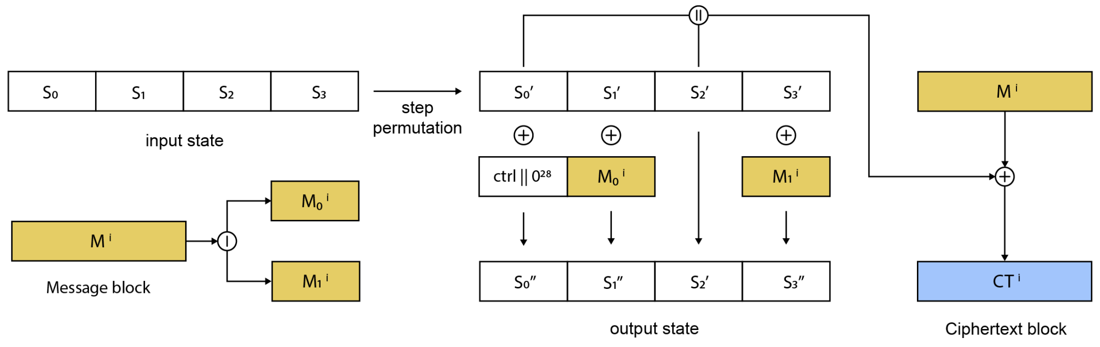

Figure 3.6: Processing plaintext during encryption

The output state of plaintext process is XORed with $ctrl||0^{252}$ and step permutation is performed. The sub states $S_1||S_3$ of the resulting state make up the 128-bit tag.

3.3 WAGE

WAGE, a lightweight cipher is based on the Welch-Gong stream cipher. This is an iterative permutation with a state size of 259 bits. A state S^i consists 37 internal states

S_j^i, $0 j 36$, each of 7-bits. Starting from the MSB, bits are arranged in the sequence of $S_0, S_1, \ldots S_{36}$ such that the LSB appears as the 7th bit of S_{36}.

State S_i is classified into rate (64 bits) and capacity (remaining 229 bits) parts. Data is loaded, processed, and returned as 64 bits. Associated data and plaintext are padded (if required) as in SpoC, are split into 64-bit blocks. So these input blocks are absorbed and the output ciphertext / plaintext blocks are obtained only at the rate bits of state. State is initialised with 128-bit key and nonce.The state update is based on 7 bit Welch-Gong permutations and lightweight 7 bit S-boxes Aagaard *et al.* (2019), rest is similar to the SpoC cipher. The following pseudo code explains the permutation, the left out functions $\omega(.)$ $SB(.)$, $WGP(.)$ are discussed in Aagaard *et al.* (2019). Considering 136-bit AD and 240-bit message, component information and hardware requirements for combinational logic are summarised in tables 3.3, 3.4.

Component	size	number
Adders	2-input, 8-bit	1332
XORs	2-input, 1-bit	140177

Table 3.3: WAGE component information

Site type	Used
IO pins	1000
Flip Flops	0
LUTs	0

Table 3.4: WAGE design information

Pseudo code 1: WAGE permutation

```
Input: $S = (S_0^0, S_0^1, ..., S_0^6, S_1^0, S_1^1, ..., S_{36}^6)$
Output: $S = \text{WAGE}permutation(S)$
Function StateUpdate($S, rc_0, rc_1$)
    for $i = 0, 1, 2, ..., 110$ do
        $fb = S_{31} \oplus S_{30} \oplus S_{26} \oplus S_{24} \oplus S_{19} \oplus S_{13} \oplus S_{12} \oplus S_8 \oplus S_6 \oplus \omega(S_0)$
        $S_5 = S_5 \oplus SB(S_8)$
        $S_{11} = S_{11} \oplus SB(S_{15})$
        $S_{19} = S_{19} \oplus WGP(S_{18}) \oplus rc_0^i$
        $S_{24} = S_{24} \oplus SB(S_{27})$
        $S_{30} = S_{30} \oplus SB(S_{34})$
        $in = fb \oplus WGP(S_{36}) \oplus rc_1^i$
        $S = \{S[7:258], in\}$
    end
return $S$
```

CHAPTER 4

Hardware implementation

Lightweight Cryptography (LWC) Hardware Application Programming Interface (API) is used to implement authenticated ciphers efficiently with several constraints like permitted widths of input, output ports, single clock throughout the core Kaps *et al.* (2019).

- PreProcessor is the unit responsible for passing input blocks to the CryptoCore and records the remaining blocks, thereby specifying the type of input data (AD, PT/CT) and their nature (partial or not) through bdi_type signal.
- CryptoCore depends on the cipher, so, this unit performs padding, processes blocks of input data, provides CT and tag during encryption, PT and verification signal during decryption.
  - Controller mainly commands the stages of algorithm (reset, key stage, npub stage, initialisation, processing AD blocks, processing PT/CT blocks, pre-tag stage, finish tag) by instructing enable signals accordingly.
  - Datapath is solely dependent on cipher, this module directly collects pdi and sdi data from PreProcessor and yields the output ciphertext, tag and passes to PostProcessor.
- PostProcessor takes care of invalid bytes of output words, and functions with respect to message authentication signal.

The input data is categorised as secret data input (sdi) and public data input (pdi). Secret key enters through sdi, and nonce, AD, PT/CT, and tag are passed through pdi and block data input (bdi).

General signals that make up the LWC core are explained below:

- **bdi, sdi** signals are of width 32 bits (4 bytes) carrying input data as mentioned above.
- **bdi_valid, sdi_valid** specify the instants when input is fed. **bdi_valid_bytes** of size, bdi_size/8 specifies the valid bytes of present input block. The signal is 1110, if 24 bits (first 3 bytes) of the current block (maximum of 4 bytes) are valid.
- **bdi_size** conveys the size of valid bytes in the current 32-bit block.
- **bdi_pad_loc** commands the padding by providing the starting location to pad. The size of this signal is same as that of bdi_vali_bytes. The signal is 0001 for the above example, so that the last byte is padded according to the cipher. For complete block inputs, bdi_valid_bytes is 1111 and bdi_pad_loc is 0000.
- **bdi_ready, sdi_ready** indicates that the module is prepared to take inputs.
- **bdi_type**, a 4-bit encoded representation identifies bdi data as nonce, AD, PT/CT, and tag.

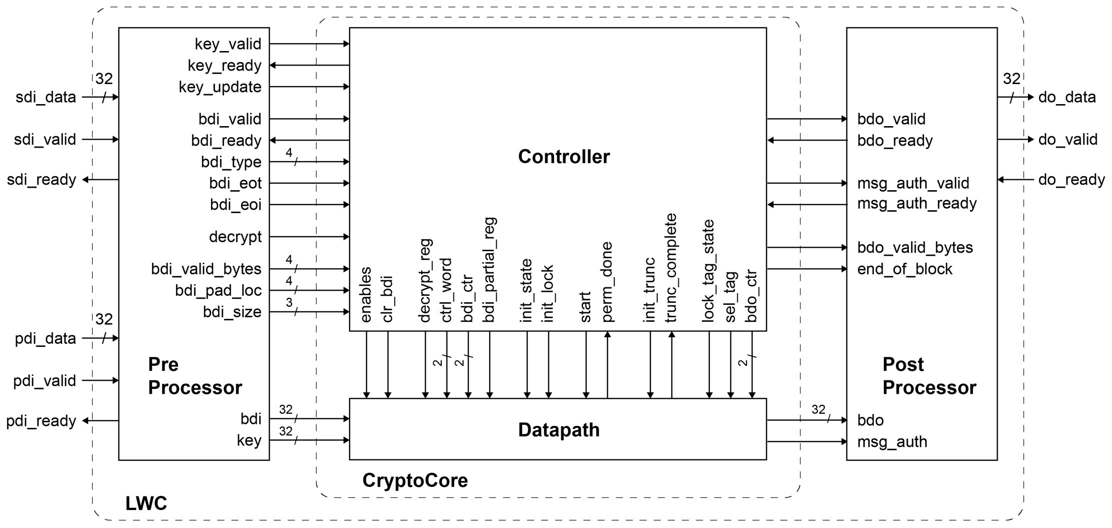

Figure 4.1: Top level block diagram of LWC core

- **bdi_eot** intimates the end of type that is being sent. This signal is reset by default, is set high to specify the last block of current input type. This signal is 1, only during the last blocks of nonce, AD and PT.
- **bdi_eoi** is set high to specify the last block of overall inputs. For example, if there exist no associated data and plaintext during encryption process, bdi_eoi is high at the last block of nonce itself, so that, the tag generation process is started.

Signals that are specific to SpoC-128 are explained below:

- **enables** is set of enable key, enable npub, enable bdi, enable state, and enable cumulative size signals which activates the corresponding activities.
- **decrypt, decrypt_reg** is low during the process of encryption and are high during decryption.
- As the key, npub sizes are of known sizes and are mandatory, these inputs can be automatically identified. So, **ctrl_word** (2 bits) indicate AD with 01 and PT/CT with 10, this representation simplifies the 4-bit control code used later, within the SpoC-128.
- **bdi_ctr** holds the number of 32-bit turns taken to process one 128-bit input block of AD/PT/CT. Similarly, **cum_size** holds the number of bytes, so, it is of $\log(128/8)$ bits.
- Initialisation stage is signalled by setting **init_state** and **init_lock** to 1.
- The step permutation of SpoC algorithm, consisting 18 rounds, is commenced by exciting **start** signal once. Datapath module, after the completion of 18 rounds, sets **perm_done** to high.
- The controller indicates truncation of partial blocks through **init_trunc**. Datapath replies the accomplishment of the task through **trunc_complete** signal.
- The tag process state is specified by setting **lock_tag_state** to 1, and **sel_tag** $= 1$ extracts tag through bdo port.

CHAPTER 5

Compression

We have now dealt with different encryption schemes that include lightweight ciphers such as WAGE and SpoC. However, in real-world problems due to technological advancements, in addition to security, integrity and authenticity, one must also be able to transmit the obtained information compactly.

Especially when working with the data dealing with signals like audio & video, our eyes and ears cannot distinguish subtle changes. These unnoticeable details can be eliminated to save up storage tremendously. We use compression techniques to eliminate this unwanted data. This lets us have shorter plain text and ciphertext. Hence, time gets reduced while encrypting, decrypting and transmitting the data.

Compression techniques are broadly classified into two categories, lossy and lossless. JPEG technique arguably comes under lossy compression, and from here on, we will see the implementation of Joint Photography Experts Group (JPEG) compression in detail.

5.1 Overview

A compression technique is effective for a sparse signal, and this requires a transformation to be applied to the signal of interest. Discrete Cosine Transform (DCT), a transform similar to Fourier transform, is utilized to discard the higher frequency content (thereby making the signal sparse), which have a less visual effect on the image. Then, we proceed by reducing large coefficients into smaller ones by quantisation. Lossless techniques (Run-length and Huffman encoding) are then applied to the quantised coefficients.

These are the typical steps involved: Color space transformation (required when dealing with colour images), splitting the image into 8x8 blocks and shifting, DCT, quantisation, zigzag, RLE, Huffman coding.

The below-discussed process is applied on every 8x8 block (here on, named a minimal coded unit, MCU), working from left to right and top to bottom. A black and white image is considered throughout the report, so the first step, colour space transformation, is explained in appendix A. A sample 8x8 block M is considered below.

$$
\mathrm{M} = \begin{bmatrix}
62 & 55 & 55 & 54 & 49 & 48 & 47 & 55 \\
62 & 57 & 54 & 52 & 48 & 47 & 48 & 53 \\
61 & 60 & 52 & 49 & 48 & 47 & 49 & 54 \\
63 & 61 & 60 & 60 & 63 & 65 & 68 & 65 \\
67 & 67 & 70 & 74 & 79 & 85 & 91 & 92 \\
82 & 95 & 101 & 106 & 114 & 115 & 112 & 117 \\
96 & 111 & 115 & 119 & 128 & 128 & 130 & 127 \\
109 & 121 & 127 & 133 & 139 & 141 & 140 & 133
\end{bmatrix}
$$

5.2 Preprocessing

The given grayscale image is first divided into non-overlapping 8x8 blocks. If the image dimensions are not divisible by 8, then the image is padded up with some dummy pixels such that the original image is not disturbed much.

Grayscale pixel intensities range from 0(black) to 255(white), and so they can be stored in 8 bits ($log_2 256$). Each pixel intensity/value is subtracted with 128, making things easier for DCT. The revised range $[-128, 127]$ is now centred around zero (similar to cosine). The processed MCU now becomes $Shifted\_M$.

$$
Shifted\_M = \begin{bmatrix}
-66 & -73 & -73 & -74 & -79 & -80 & -81 & -73 \\
-66 & -71 & -74 & -76 & -80 & -81 & -80 & -75 \\
-67 & -68 & -76 & -79 & -80 & -81 & -79 & -74 \\
-65 & -67 & -68 & -68 & -65 & -63 & -60 & -63 \\
-61 & -61 & -58 & -54 & -49 & -43 & -37 & -36 \\
-46 & -33 & -27 & -22 & -14 & -13 & -16 & -11 \\
-32 & -17 & -13 & -9 & 0 & 0 & 2 & -1 \\
-19 & -7 & -1 & 5 & 11 & 13 & 12 & 5
\end{bmatrix}
$$

5.3 Discrete Cosine Transform

DCT is a cut-down version (only the real part) of the Fast Fourier Transform, so the produced signal size (N samples) is the same as input size N. The value N in the N-length DCT is decided as 8 in JPEG. The idea of 1D-DCT is applied row-wise, then column-wise on the image to obtain the required 2D-DCT. DCT represents the finite-length sequence as a summation of cosine terms of different frequencies. So the result of the forward DCT applied matrix is the 64-coefficients in the frequency domain. The formula for 2D-DCT is given by eq.5.1.

$$D(i,j) = \frac{1}{4}C(i)C(j)\sum_{x=0}^{7}\sum_{y=0}^{7} p(x,y) cos\left[\frac{(2x+1)i\pi}{16}\right] cos\left[\frac{(2y+1)i\pi}{16}\right] \tag{5.1}$$

$$\text{where, C(u)} = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u = 0 \\ 1 & \text{if } u \geq 0 \end{cases}$$

In order to perform the same in Verilog, the expression is manipulated as a matrix, evidently of 8x8 dimension. The expression for 1D-DCT, $N = 8$ is considered in eq.5.2 and the same is made up as eq.5.5.

5.3.1 1D DCT

$$F(u) = \alpha(u)\sum_{i=0}^{7} f(i)\ \ cos\left[\pi u\frac{(2i+1)}{16}\right], \quad \alpha(u) = \begin{cases} \sqrt{\frac{1}{8}} & \text{if } u = 0 \\ \sqrt{\frac{2}{8}} & \text{if } u \geq 0 \end{cases} \tag{5.2}$$

The input f(i) and output F(u) are 1x8 matrices. As all the cosine values can be predetermined for $u = 0$ to 7, the value of F(2) is found in eq.5.3 and is modified as eq.5.4.

$$F(2) = \frac{1}{2}\sum_{i=0}^{7} f(i)\ \ cos\left[\pi\frac{(2i+1)}{8}\right] \tag{5.3}$$

$$F(2) = \left[f(0) \;\; f(1) \;\; ... \;\; f(7) \right] * \left[cos(\frac{\pi}{8}) cos(\frac{3\pi}{8}) ... cos(\frac{15\pi}{8}) \right]^{T}_{1x8} \tag{5.4}$$

Similarly, each $F(u)$ is manipulated as $[inputs] * [fixed\ values]^{T}_{8x1}$, so that each such column of fixed values gets concatenated to form the final fixed matrix (say T_{8x8}). Assuming the input/image matrix as M, we now know that 1D-DCT is same as $M_{1x8} * T_{8x8}$.

$$\left[F(0) \;\; ... \;\; F(7) \right] = \left[f(0) \;\; f(1) \;\; ... \;\; f(7) \right] * \left[fixed\ matrix \right]_{8x8} \tag{5.5}$$

5.3.2 2D DCT

Extending the similar idea to the 2D-DCT gives the fixed matrix T as shown below, and the required DCT output as per eq.5.1 is same as $T * M * T^{T}$ or $D = TMT'$.

$$T = \begin{bmatrix} 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 \\ 0.4904 & 0.4157 & 0.2778 & 0.0975 & -0.0975 & -0.2778 & -0.4157 & -0.4904 \\ 0.4619 & 0.1913 & -0.1913 & -0.4619 & -0.4619 & -0.1913 & 0.1913 & 0.4619 \\ 0.4157 & -0.0975 & -0.4904 & -0.2778 & 0.2778 & 0.4904 & 0.0975 & -0.4157 \\ 0.3536 & -0.3536 & -0.3536 & 0.3536 & 0.3536 & -0.3536 & -0.3536 & 0.3536 \\ 0.2778 & -0.4904 & 0.0975 & 0.4157 & -0.4157 & -0.0975 & 0.4904 & -0.2778 \\ 0.1913 & -0.4619 & 0.4619 & -0.1913 & -0.1913 & 0.4619 & -0.4619 & 0.1913 \\ 0.0975 & -0.2778 & 0.4157 & -0.4904 & 0.4904 & -0.4157 & 0.2778 & -0.0975 \end{bmatrix}$$

The columns of T form an orthonormal set, so T is an orthogonal matrix. This property becomes useful while performing inverse DCT as the inverse of T is T'. As all the values in T turn out to be less than 0.5 in magnitude, the sign (1 bit) and the mantissa (say, 20 bits) are only stored in verilog.

$$D = TMT' = \begin{bmatrix} -242 & 98.3 & 125.4 & 125.5 & 126.9 & 124.3 & 126.5 & 127.92 \\ -103 & 172.9 & 152.5 & 127.7 & 137.3 & 131.9 & 132.3 & 126.6 \\ 190.8 & 136.5 & 120.4 & 125.3 & 128.3 & 127.6 & 128.5 & 127.2 \\ 140.5 & 113.4 & 124.5 & 124.6 & 130.4 & 126.7 & 130.7 & 127.6 \\ 123.1 & 124.1 & 128.9 & 131.6 & 128.1 & 133.1 & 129.1 & 128.5 \\ 127.5 & 131.1 & 126.6 & 128.2 & 126.9 & 126.5 & 126.9 & 128.9 \\ 132.4 & 130.3 & 126.3 & 126.4 & 129.1 & 125.3 & 129.1 & 126.6 \\ 117.8 & 126.2 & 133.9 & 127.6 & 128.3 & 128.4 & 127 & 128 \end{bmatrix}$$

The important note here is that the image pixel values are supplied sequentially (1 pixel at a time) by the camera module, as it has only eight pins for transmission. Pixel values are sent in a row-wise fashion, starting with the top-left pixel, and this is the convention for most cameras. For an image of 256x480, it is stressed that after obtaining the first row of the 1st MCU, the first row of the 2nd starts entering. Similarly, MCU 3 gets all of its entries 8 cycles after the completion of MCU 2. First element of MCU 33 enters after all the elements of MCUs 1-32 are obtained. Partial computations MT' and TMT' are started as soon as a new element enters the MCU instead of waiting till its completion.

MCU 1	MCU 2	MCU3	...	MCU 32
MCU 33	MCU 34	MCU35	...	MCU 46
...	...	...	...	...
MCU 1889	MCU 1890	MCU 1891	...	MCU 1920

Since each MCU holds its values row-wise, MT' is performed first, making the matrix multiplication simpler than calculating TM first. Matrix 'M' is the message/image of 8-bit intensities. T is the fixed matrix of 20-bit mantissa, so each element of MT' matrix is of 31-bits ($20+8+log_2 8$, as 8 products are added), and the decimal point is after 20 bits from the LSB. Similarly, TMT' matrix capacity should be 54-bits ($31+20+log_2 8$), and the decimal point is after 40 bits from the LSB.

5.3.3 Importance and significance

The DCT output matrix represents the magnitudes of frequency variations of increasing order. The fig.5.1 shows the 64 bases of 8-sample 2D-DCT, and each base is again a 64-pixel configuration. It can be observed that the top-left configuration is plain, and it holds the dc information (average) of the MCU, and all the remaining bases are ac components.

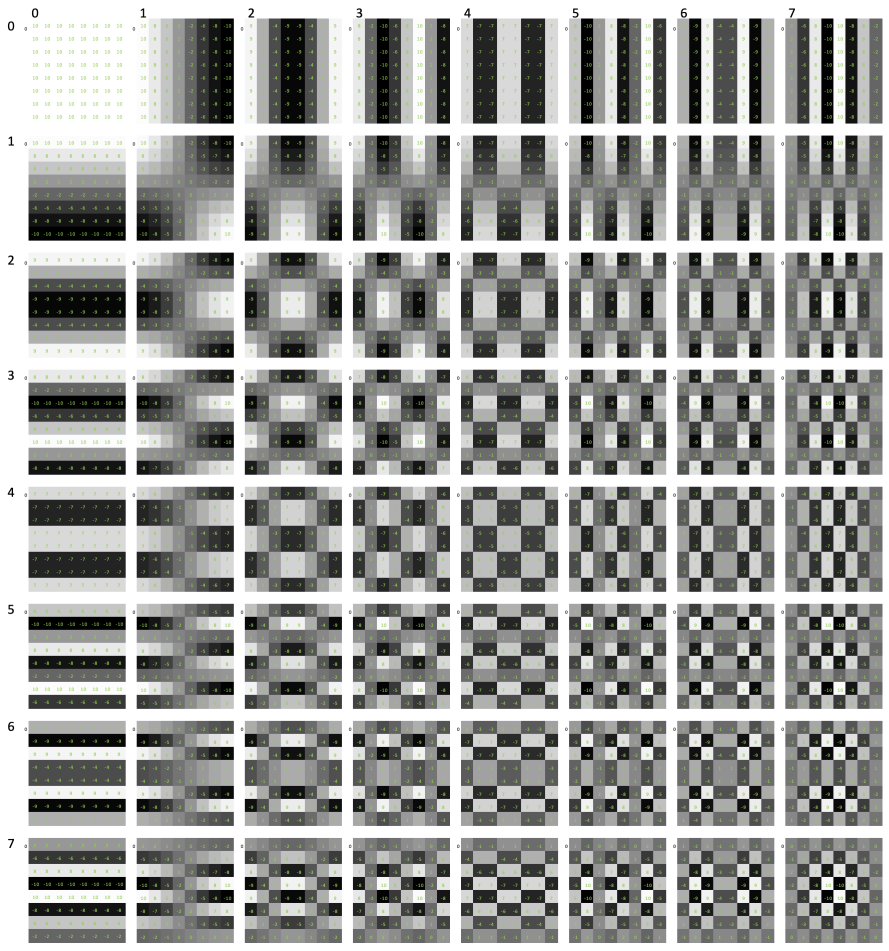

Figure 5.1: Bases of 2D-DCT

These resultant 64 coefficients are the weights of the 64 configurations with different frequency variations as shown in the fig.5.1. The actual 8x8 image chunk can be represented as a weighted combination of all these configurations. In simple words, the forward DCT can be understood as each output value capturing the similarity of the actual MCU with its corresponding configuration. So, the first row elements of the DCT output represent the x-axis variation of the 8x8 image.

One of the advantages of using this DCT process is that real-world images often have very gradual brightness and colour changes, leading to smaller high-frequency coefficients of the DCT output. This step is lossless and so is reversible. The matrix D can be manipulated in the Quantisation section of the JPEG algorithm.

5.4 Quantisation

This is the lossy segment of the JPEG algorithm. This step can be achieved by dividing the DCT output element-wise with the quantisation matrix. This denominator matrix depends on quality values ranging from 1 to 100, 1 being the poorest image quality and highest image compression, 100 being the best image quality and lowest compression. Quality level 50 is used here, which gives both high quality and excellent image compression.

$$Quantisation_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Generally, images tend to have low-frequency variation changes in a localized area. Combined with the tendency for humans to not notice high-frequency changes, a large amount of the high order frequency variation coefficients are discarded. This is implemented by choosing the quantisation factors appropriately for the coefficient to be quantised. For example, the dc coefficient data is important to retain, so a small quantisation factor is chosen for it. This value is around 10 to 16. Conversely, the highest order ac coefficient (8,8) is not essential to retain, so a large quantisation factor is chosen. This number can range between 80 to 140. The other ac values are in between the two extremes. quantisation is achieved by dividing each DCT output element by the corresponding element in the Q matrix and then rounding to the nearest integer value. The possible range of DCT output values are in between [-1024, 1023], so if we consider the least value in the quantisation matrix (i.e, 10), we get quantised output values in the range (-103, 103) which can be represented using 8-bits.

$$Q = D./Quantisation_{50} = \begin{bmatrix} -23 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -21 & 4 & 2 & 0 & 0 & 0 & 0 & 0 \\ 6 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

5.5 Zigzag and Run-length encoding

The run-length encoding algorithm is lossless and effective when the data sequence is consecutively repetitive (only repetitive zeros here). The quantised matrix, however, contains many zeros in the bottom right quadrant. To make the Q matrix suitable for run length scheme, it cannot be unrolled in row-wise or column-wise tradition as the subject of compression (zeros) is split into smaller chains. So, the zigzag fashion is adapted for the Q matrix before applying RLE, illustrated in the appendix.

RLE represents the compressed data as sequence of terms (x,y), and the 1st entry of each term (called frequency) conveys the number of zeros preceding the second term. So, 'x' zeros followed mostly by a non-zero element 'y' are compressed as (x,y). For example, data 23, 0, 5, 0, 0, 0, 0, 1 is stored in 40 bits (i.e, 5 bits per element). The RLE output of the data is (0,23), (1,5), (4,1) requiring 27 ($=$ (3x4) $+$ (3x5)) bits only. 4 bits should only be allocated to the frequency, implying that $2^4 - 1$ (= 15) preceding zeros can only be stored by each term. A typical exception is that the string of 20 zeros are stored as (15,0),(3,0). The first term holds 15 preceding zeros followed by a zero. The second similarly holds four zeros, noting that 'y' ought to be zero only in cases of greater than 15 consecutive zeros or at the end.

The example MCU is now shortened as,
$Z = (0,-23),(0,-2),(0,-21),(0,6),(0,4),(2,2),(0,1),(0,1),(1,-1),(15,0),$
$(15,0),(15,0),(3,0)$

5.6 Huffman coding

- **Variable length coding**: This method maps elements to binary code of variable length. In contrast to 4 = 100, 2 = 010, the code length is not fixed in VL coding. Here 0,1,2,3,4 can be coded as 0|1|00|01|10 and so on. The encoding is simple, and the values can be transferred efficiently. The challenge occurs at the decoding end, where the signal string 0100 can be misunderstood as 01|0|0.
- To fix the issue, **prefix-free codes** came into existence. These codes cannot be misunderstood as any of the code prefixes does not match. The same 0,1,2,3,4 are now 0|10|110|1110|1111, the decoder cannot be misled. The signal cannot be understood as 010|1|10 because these bit strings are not assigned to any of the elements. The downside in this approach is also evident. The last couple of elements require 4 bits, i.e. greater than the conventional 3-bit fixed-length coding.

Huffman coding generally is employed for encoding information as prefix-free codes. To overcome the limitation of prefix-free codes (PF codes), the code with the smallest length is assigned to the most frequent element. The longest code represents the least repeated element, thereby drawing an overall advantage. The similar idea is observed in the Morse code where the frequent letter 'e' is coded as *dot* and 'y' as *dash dash dot dash*.

The same Huffman coding is adapted now with some tweaks. The smaller values 0,-1,1,-2,2,... are presumed as the most frequent elements. As the magnitude increases, their occurrence drops usually. The complication of PF codes is magnified now. As the number of elements to be encoded increases, the code lengths for many elements rise tremendously. The previous 5 numbered example required maximum of 4 bits to represent an element. A set of 100 elements may require maximum of 60 bits per element and is much higher than using 7-bit fixed-length coding. So, the number of elements in the symbol table should be limited.

The possible elements are, however, fixed and cannot be avoided. This led to the segregation of all possible elements into families. Each element falls into a family, and the capacity of each family is again decided thoughtfully. The possible 207 elements (range of Q matrix, [-103,103]) are categorised as seven families, so the symbol table of the PF method is reduced significantly. The Prefix-free code now indicates just the family but cannot identify the element in it. Based on the number of elements in each family, a fixed-length code is set up to find the required element.

The dc value of an MCU is comparable to adjacent MCU dc values, and so we can use this advantage in its encoding process. The dc value of the previous MCU is subtracted from dc value of the present MCU.

This difference or value, finds its corresponding *code_R* from column-2 of table 5.1. Prefix-free *code_L* is also obtained from table 5.1, and $\{code\_L, code\_R\}$ gives the encoded dc value. However, for ac values, encoding of each term (x,y) is proceeded by choosing the family and corresponding *code_R* of value 'y' from table 5.1. Let 'y' be mapped to family 'f'.

Now, x/f chooses the *code_L* from the table 5.2. 'x' is the run and 'f' is the family, and also indicates the length of *code_R*. $\{code\_L, code\_R\}$ gives the encoded ac value. It must be noted that the end of input (EOI) is denoted by 1010. All of the trailing zeros after zigzag, can simply be replaced with 1010.

Value (DC difference or y)	Code_R (for both dc & ac terms)	Family 'f'	Code_L (for dc)
0	-	0	00
-1, 1	0, 1	1	010
-3, -2, 2, 3	00, 01, 10, 11	2	011
-7, -6, -5, -4, 4, 5, 6, 7	000, 001, 010, 011, 100, 101, 110, 111	3	100
-15, ... , -8, 8, ... , 15	0000, ... , 0111, 1000, ... ,1111	4	101
-31, ... , -16, 16 ... , 31	00000, ... , 01111, 10000, ... , 11111	5	110
-63, ... , -32, 32 ... , 63	000000, ... , 011111, 100000, ... , 1111111	6	1110
-103, ... , -64, 64, ... , 103	0011000, ... , 0111111, 1000000, ... , 1100111	7	11110

Table 5.1: Family table

The RLE output Z = (0,-23), (0,-2), (0,-21), (0,6), (0,4), (2,2), (0,1), (0,1), (1,-1), (15,0), (15,0), (15,0), (3,0) is encoded as shown.

- DC term: DC value of present MCU is -23, let the previous dc value (preferably left MCU) be -16. The difference is -7 (= -23 -(-16)). -7 maps to family 3 and its corresponding *code_R* is 000. The family code for f = 3 is *code_L* = 100. So, the dc encoded output is **100000**.

- AC terms: The term (0,-2) is considered. Here x = 0 and y = -2. y = -2 maps to f = 2 and *code_R* is 01. x/f is 0/2, so, *code_L* is 01. The encoded output of (0,-2) is just **0101**. The similar process is continued till the term (1,-1).

- The trailing zero terms (15,0), (15,0), (15,0), (3,0) are jointly encoded as **1010**. During the decoding stage, if the string 1010 is observed, all the previously decoded terms (x,y) are expanded as x zeros and y. Count of the previous elements reveal the count of trailing zeros, as total elements are 64. The remaining zeros are found to be 52 and there is a single way to represent these 16+16+16+4 zeros i.e. (15,0), (15,0), (15,0), (3,0).

- We again start with the dc difference while decoding the compressed MCU **1000000101...1001**. It is to be noted that the decoding starts with finding suitable *code_L* which matches the beginning of the output string. Observing *code_L* from table 5.1, prefix of the output bit string **1000000101...1001** matches only with 100, confirming the family 3. f = 3 further indicates that only the succeeding 3 bits of '100' belong to the dc term. These 3 bits, 000 matches to **-7** in the family 3. As the first MCU yields its dc value (not the difference), each MCU's dc value can be eventually calculated by their difference.

- The output is now reduced to **0101...1001** and we are left with ac terms. The prefix of **0101...1001** matches only to **01** of table 5.2, implying x=0, f=2. Family 2 reveals that the following two bits i.e. **01** falls under the same term and this value maps y to -2. The same method is followed till the output is reduced to **1010**.

x/f	Code word	x/f	Code word	x/f	Code word
0/1	00	5/1	1111010	10/1	111111010
0/2	01	5/2	11111110111	10/2	1111111111000111
0/3	100	5/3	1111111110011110	10/3	1111111111001000
0/4	1011	5/4	1111111110011111	10/4	1111111111001001
0/5	11010	5/5	1111111110100000	10/5	1111111111001010
0/6	1111000	5/6	1111111110100001	10/6	1111111111001011
0/7	11111000	5/7	1111111110100010	10/7	1111111111001100
1/1	1100	6/1	1111011	11/1	1111111001
1/2	11011	6/2	111111110110	11/2	1111111111010000
1/3	1111001	6/3	1111111110100110	11/3	1111111111010001
1/4	111110110	6/4	1111111110100111	11/4	1111111111010010
1/5	11111110110	6/5	1111111110101000	11/5	1111111111010011
1/6	1111111110000100	6/6	1111111110101001	11/6	1111111111010100
1/7	1111111110000101	6/7	1111111110101010	11/7	1111111111010101
2/1	11100	7/1	11111010	12/1	1111111010
2/2	11111001	7/2	111111110111	12/2	1111111111011001
2/3	1111110111	7/3	1111111110101110	12/3	1111111111011010
2/4	111111110100	7/4	1111111110101111	12/4	1111111111011011
2/5	1111111110001001	7/5	1111111110110000	12/5	1111111111011100
2/6	1111111110001010	7/6	1111111110110001	12/6	1111111111011101
2/7	1111111110001011	7/7	1111111110110010	12/7	1111111111011110
3/1	111010	8/1	111111000	13/1	11111111000
3/2	111110111	8/2	111111111000000	13/2	1111111111100010
3/3	111111110101	8/3	1111111110110110	13/3	1111111111100011
3/4	1111111110001111	8/4	1111111110110111	13/4	1111111111100100
3/5	1111111110010000	8/5	1111111110111000	13/5	1111111111100101
3/6	1111111110010001	8/6	1111111110111001	13/6	1111111111100110
3/7	1111111110010010	8/7	1111111110111010	13/7	1111111111100111
4/1	111011	9/1	111111001	14/1	1111111111101011
4/2	1111111000	9/2	1111111110111110	14/2	1111111111101100
4/3	1111111110010110	9/3	1111111110111111	14/3	1111111111101101
4/4	1111111110010111	9/4	1111111111000000	14/4	1111111111101110
4/5	1111111110011000	9/5	1111111111000001	14/5	1111111111101111
4/6	1111111110011001	9/6	1111111111000010	14/6	1111111111110000
4/7	1111111110011010	9/7	1111111111000011	14/7	1111111111110001
15/0	11111111001	15/1	1111111111110101	15/2	1111111111110110
15/3	1111111111110111	15/4	1111111111111000	15/5	1111111111111001
15/6	1111111111111010	15/7	1111111111111011	EOI	1010

Table 5.2: Prefix-free *code_L* for ac terms

CHAPTER 6

Results and Discussions

Our study resulted in a tangible output as we have developed a hardware code capable of implementing the discussed AE algorithms. The $1,088$ test vectors (comprising several combinations of associated data and plaintext, for fixed 128-bit key and nonce) of SpoC-128 provided with the official C-language code were passed successfully by our hardware code. The hardware requirements of **ZYNQ-7 ZC702 Evaluation Board** for the proposed HDL code are indicated in tables 6.1, 6.2.

Module	component	size	number
Controller	Adders	2-input, 5 bit	1
		2-input, 2 bit	3
	Registers	1-bit	11
	Muxes	2-input, 1-bit	1020
Datapath	Adders	2-input, 5 bit	3
	XORs	2-input, 1-bit	261
	Registers	1-bit	391
	Muxes	2-input, 1-bit	3860
	LUTs	32x8	4

Table 6.1: Cryptocore component information

Site type	Used	Utilised(%)
IO pins	105	52.5
Flip Flops	919	0.86
LUTs	2711	5.1
IO Buffers	105	-

Table 6.2: Utilisation design information

The future scope will be to test these on FPGA board and implementing Dilithium, a post quantum cryptography algorithm.

CHAPTER 7

SUMMARY AND CONCLUSION

Till date, we have completed a detailed study of the state of the art algorithms namely Ascon, ACORN, WAGE, and SpoC. Additionally, we have successfully completed the hardware code for ACORN and WAGE. SpoC, which can be used for power & area constrained applications is modified and simulated such that the algorithm can directly be put to practical use. The cipher is made compatible with the discussed LWC API such that the core intakes, processes, and produces fixed length of bit strings.

We, in detail studied JPEG compression and also wrote the hardware code for the same in verilog. The code was written for grayscale images, however, can be used for colour images by including colour transformation as mentioned in the appendix A. Encryption along with compression technique can be much more effective than encryption alone as transmission time and data storage reduces.

REFERENCES

1. (2020). *Encryption and Authentication Algorithms*. https://sourcedaddy.com/networking/encryption-and-authentication-algorithms.html.

2. (2020). *Origin of Cryptography*. https://www.tutorialspoint.com/cryptography/index.htm.

3. **M. Aagaard**, **R. AlTawy**, **G. Gong**, **K. Mandal**, **R. Rohit**, and **N. Zidaric** (2019). Wage: An authenticated cipher.

4. **A. Ahuja** (2020). *Quantum Cryptography*. https://www.geeksforgeeks.org/quantum-cryptography/?ref=rp.

5. **R. AlTawy**, **G. Gong**, **M. He**, **A. Jha**, **K. Mandal**, **M. Nandi**, and **R. Rohit** (2019). Spoc. *Submission to NIST LwC Standardization Process (Round 2)*.

6. **C. A.R.** (2016). *Authenticated encryption*. http://cryptowiki.net/index.php?title=Authenticated_encryption.

7. **article** (). *Huffman Coding Algorithm*. https://www.studytonight.com/data-structures/huffman-coding.

8. **K. Cabeen** and **P. Gent** (2008). Image compression and discrete cosine transform‖, college of redwoods.

9. **A. D'Angelo** (). *Discrete Cosine Transform*. http://users.dimi.uniud.it/~antonio.dangelo/MMS/2013/lessons/L11lecture.pdf.

10. **S. Goldwasser** and **M. Bellare** (). Lecture notes on cryptography. *Summer course "Cryptography and computer security" at MIT*.

11. **J. W. Jehhal Liu** (). Jpeg compression and ethernet communication on an fpga.

12. **Juniper** (2020). *Authentication Algorithms*. https://www.juniper.net/documentation/en_US/junos/topics/concept/ipsec-authentication-solutions.html.

13. **J.-P. Kaps**, **W. Diehl**, **M. Tempelmeier**, **E. Homsirikamol**, and **K. Gaj** (2019). Hardware api for lightweight cryptography. *URL https://cryptography. gmu. edu/athena/index. php*, 1–26.

14. **J. Kothari** (2020). *Cryptography and its Types*. https://auth.geeksforgeeks.org/user/JASHKOTHARI1/articles.

15. **D. McGrew** (2008). An interface and algorithms for authenticated encryption. Technical report, RFC 5116, January.

16. **D. Mukhopadhyay** (2014). Cryptography and network security.

17. **F. Piper** and **S. Murphy**, *Cryptography: A Very Short Introduction*. Oxford University Press, 2013.

18. **P. Reddy** (2019). *Real Life Applications of Cryptography*. https://medium.com/@prashanthreddyt1234/real-life-applications-of-cryptography-162ddf2e917d.

19. **Rhea** (). Jpeg compression.

20. **V. Rijmen** (2011). *Authenticated Encryption*. https://www.cryptomathic.com/news-events/blog/authenticated-encryption.

21. **G. Singh** (2019). *Development of Cryptography*. https://www.geeksforgeeks.org/development-of-cryptography/?ref=rp.

22. **W. Stallings**, *Cryptography and network security, 4/E*. Pearson Education India, 2006.

23. **I.-T. T.81** (1993). *INFORMATION TECHNOLOGY – DIGITAL COMPRESSION AND CODING*. https://www.w3.org/Graphics/JPEG/itu-t81.pdf.

24. **D. M. Turner** (2019). *Summary of cryptographic algorithms - according to NIST*. https://www.cryptomathic.com/news-events/blog/summary-of-cryptographic-algorithms-according-to-nist.

25. **H. Wu** (2016). Acorn: a lightweight authenticated cipher (v3). *Candidate for the CAESAR Competition. See also https://competitions. cr. yp. to/round3/acornv3. pdf*.

APPENDIX A

Colour space

The Human Visual System describes the way that the human eye processes an image. The human eye is more sensitive to the luminance(brightness) than the chrominance(colour difference) of an image, there by requiring another set of quantisation table and Huffman tables for chrominances. Thus during compression, chrominance values are less important, and quantization can be used to reduce the amount of psycho-visual redundancy. This promotes us to adapt to the Y (luminance),Cb (blue chrominance),Cr (red chrominance) representation of pixels. If interested, Cg (green chrominance) can be calculated from Cb and Cr.

$$Y = 16 + \frac{65.738R}{256} + \frac{129.057G}{256} + \frac{25.064B}{256} \tag{A.1}$$

$$Cb = 128 - \frac{37.945R}{256} - \frac{74.494G}{256} + \frac{112.439B}{256} \tag{A.2}$$

$$Cr = 128 + \frac{112.439R}{256} - \frac{94.154G}{256} - \frac{18.285B}{256} \tag{A.3}$$

APPENDIX B

Zigzag

The zigzag fashion that is adapted for the quantised matrix (i.e. Q matrix) before applying run-length encoding is illustrated in the fig.B.1.

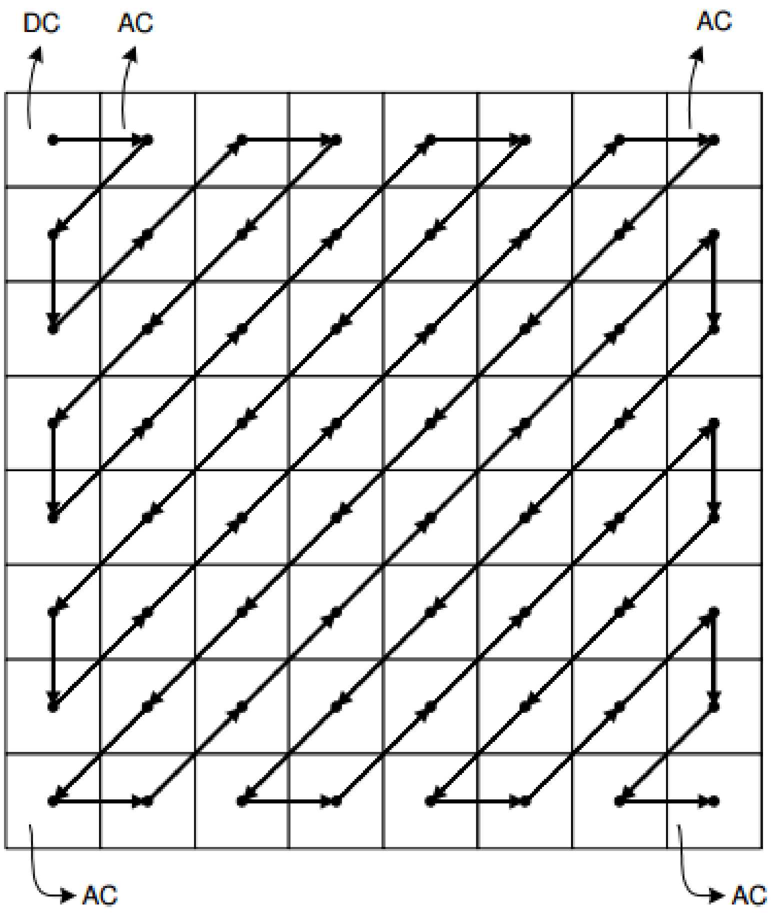

Figure B.1: The zigzag sequence

APPENDIX C

Verilog implementation

Implementation of JPEG is elaborated here. A module is created for each MCU with inputs I, rst, clk, enable, prev_dc_value, start_MCU and first_MCU, and the outputs being ready, output, done, present_dc_value, Valid_output.

- Input pixel values enter sequentially, in row-wise fashion. The first row of an MCU arrives in 8 consecutive cycles and the next row starts entering after n-8 cycles for an mxn image. Each input is an 8-bit pixel intensity. This input is stored by signal $[7:0]\mathbf{I}$.
- **rst** (reset) and **clk** (clock) are the single bit inputs. **start_MCU** signal is low only if the MCU is empty.
- The signal **first_MCU** is required for the starting MCU, as there will be no previous dc value to be subtracted before encoding its dc value. The signal is high only for MCU 1.
- The **enable** signal is required to differentiate the sets of 8-cycles during which the input is valid. The **enable** is high during the valid inputs.
- **prev_dc_value** is necessary at the beginning of Huffman encoding, it should be subtracted from the **present_dc_value** (which should be sent out, for the succeeding MCU). These signals are of 14-bits. The range of these signals depend on the quantisation matrix and fall under [-1024, 1023], if not quantised.
- Signal **ready** is low only if all the inputs are available in the MCU. Signal **done** is excited after completion of all the sequential outputs.
- Huffman encoded bit strings of every (x,y) term are represented by the 23-bit **output** signal (the longest possible code for any ac or dc term is of 23 bits). For smaller outputs, the signal is padded with zeros on the right. Note that the signal starts with prefix-free code, which also decides the number of valid bits that stand next to this prefix code, thereby avoiding the ambiguity of the padded signal.

- The sequential output ends with bits 1010 (and zeros padded). The instances at which the sequential output is appropriate, is suggested by the active-high $\mathbf{Valid}_{output signal}$.

C.1 DCT

Applying DCT for an 8x8 matrix is same as calculating TMT'. M is the MCU and T being pre-computed, fixed matrix. As the values of M fall in row-wise fashion, MT' (also referred as *Acc* matrix) is calculated initially.

C.1.1 Determining M.T'

Element M(i,j) should interact with every element in row j of matrix T'. As soon as a single element enters matrix M, its contribution in the *Acc* matrix is updated. For this, a counter (signal *ctr*) is kept to track present input's index and this counter accordingly picks the corresponding j^{th} row elements of T' (done by signal $m1A$). These eight elements are multiplied with the present input. The eight partial products are accumulated at i^{th} row of the *acc* matrix by signal $acc1\_loc$.

Matrices M and T are of dimension 8x8 and with each location storing more than 1-bit, these matrices become 3-dimensional which is not synthesizable. So the matrices are reshaped to 1x64 while dealing in verilog.

Signals $m1A$, $m2A$, $acc1\_loc$, $acc2\_loc$, acc_loc are sets of eight signals. Set $m1A[k] = \{k, ctr[2:0]\}$, $k = 0$ to 7. The signal set $acc1\_loc$ can be realised by $acc1\_loc[k] = \{i, k\}$, i is the 3-bit input row.

Mantissa of matrix T' and their signs are stored separately. We stored the mantissa in 20 bits, but 12 or 16 bits also work fine. Input I of matrix M is of 8 bits including sign, its magnitude is used for multiplication. The 8x20 bit multiplier is unsigned and will produce 28-bit products (but these are partial-products for *Acc*, stored in signal *PP*). The capacity of each *Acc* element should be 31 bits as there are 8 partial products to be added. *Acc* matrix is obtained row-wise, is completed at the immediate cycle after all the inputs are acquired.

C.1.2 Determining T.M.T'

Evaluation of matrix TMT' (also referred as Acc2) can be started only after obtaining a complete row of inputs, before which we are left with incomplete *Acc* elements. Note that $Acc2 = T.Acc$ and *Acc* doesn't have a single entire column before its completion. So $Acc2$ values develop gradually and a single value of it cannot be decided without entire matrix M.

Similarly, the signal set $m2A$ picks the elements of matrix T. acc_loc gathers the appropriate *Acc* term. These eight products are stored in $PP2$. The partial products update the proper indices of $Acc2$ with the help of signal $acc2\_loc$. The multiplication is now 31-bit x 20-bit. Each signal of $PP2$ is of size 51 bits and $Acc2$ elements are of 54 bits $(51 + log_2 8)$ each. $Acc2$ is obtained after 71 cycles i.e. 8 cycles after obtaining all the inputs.

C.2 Quantisation

The quantisation scale is decided as 50, so the denominator matrix is fixed. Instead of performing division, the mantissa of the reciprocals of Q_{50} elements are stored in a wire and multiplied to the $Acc2$ matrix element-wise. All the values in quantisation matrix are positive, so there is no intervention of sign. The decimal point of $Acc2$ elements is after 40 bits from LSB. The mantissa of quantisation matrix are of 20-bits. The products are rounded off using the 60^{th} element from LSB. We are left with matrix Q with each element of 14 bits. The quantisation of every column is performed after 64 cycles i.e. after obtaining complete columns of $Acc2$ and is completed after 72 cycles. Signal $After\_ZZ$ holds the re-arranged elements according to the zig-zag pattern.

C.3 Run-length encoding

Run-length encoding is challenging to implement with combination logic. So a couple of quantised outputs are considered every time as shown in the pseudo code. The signal set $[11:0]RL[0:63]$ holds the run-length output terms (x,y) as {4-bit x, 8-bit y}. As shown in the pseudo code, maximum of 64 RLs are required but are generally of indefinite size j. Huffman coding is started as soon as the RLE is done. The *valid* signal is excited and the outputs follow the look-up table and the padding is done accordingly.

Pseudocode 1: Run-length encoding logic

```
j = 0, index=0
for $i = 0, 2, ..., 62$ do
    in1 = After_ZZ[i], in2 = After_ZZ[i+1]
    if in1 != 14'd0  in2 != 14'd0 then
        RL[j] = {index, in1[6:13]} // Run-length terms
        RL[j+1] = {4'd0, in2[6:13]}
        // Both non-zero inputs=> 2 new terms
        j = j+2
        index = 6'd0 // frequency reset
        last_non_zero_terms <= last_non_zero_terms + 2 + zero_terms
        zero_terms <= 7'd0 // To track end of input
    if in1 == 14'd0  in2 != 14'd0 then
        if index < 15 then
            RL[j] = {index+1'b1, in2[6:13]}
            j = j+1, index = 6'd0
            last_non_zero_terms <= last_non_zero_terms + 1 + zero_terms
            zero_terms <= 7'd0
        else
            RL[j] = {4'b1111, 8'd0}
            RL[j+1] = {4'b0000, in2[6:13]}
            //[6:13] bits are enough as the range is [-103,103]
            j = j+2, index = 6'd0
            last_non_zero_terms <= last_non_zero_terms + 2 + zero_terms
            zero_terms <= 7'd0
    if in1 != 14'd0  in2 == 14'd0 then
        RL[j] = {index, in1[6:13]}
        j = j+1, index = 6'd1
        last_non_zero_terms <= last_non_zero_terms + 1 + zero_terms
        zero_terms <= 7'd0
    if in1 == 14'd0  in2 == 14'd0 then
        if index < 6'd14 then
            index = index + 2'd2
            if i == 62 then
                RL[j] = {index-1'b1, 8'd0}
                // End of input
                j = j+1
                stop_RLE = 1'b1
                zero_terms = zero_terms + 1
        if index == 6'd14 then
            RL[j] = {4'b1111, 8'd0}
            j = j+1, index = 6'd0
            zero_terms = zero_terms + 1
        if index == 6'd15 then
            RL[j] = {4'b1111, 8'd0}
            j = j+1, index = 6'd1
            zero_terms <= zero_terms + 1
```