LIGHTWEIGHT AUTHENTICATED ENCRYPTION
FOR FPGAS

by

Upendarreddy Mamidi
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

_____	Dr. Jens-Peter Kaps, Thesis Director
_____	Dr. Kris Gaj, Committee Member
_____	Dr. Alok Berry, Committee Member
_____	Dr. Monson H. Hayes, Chairman, Department of Electrical and Computer Engineering
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Spring Semester 2016
George Mason University
Fairfax, VA

Lightweight Authenticated Encryption for FPGAs

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Upendarreddy Mamidi
Bachelor of Technology
VIT University, India, 2013

Director: Dr. Jens-Peter Kaps, Associate Professor
Department of Electrical and Computer Engineering

Spring Semester 2016
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my parents Siddiram Reddy and Manemma, brother Mahendar and sister Srilatha and beloved friends.

Acknowledgments

I would like to express my gratitude to my advisor Dr. Jens-Peter Kaps for helping me with useful comments and remarks throughout the course of the thesis. Furthermore I would like to thank Dr. Kris Gaj, Pansayya Yalla and Ekawat Homsirikamol for their continuous support in getting the work done. Finally, I would like to thank Sangamitra without whose help, it would have been difficult to complete the thesis.

Table of Contents

List of Tables

List of Figures

Abstract

LIGHTWEIGHT AUTHENTICATED ENCRYPTION FOR FPGAS

Upendarreddy Mamidi, M.S.

George Mason University, 2016

Thesis Director: Dr. Jens-Peter Kaps

Traditionally, authenticated encryption was achieved by using two seperate algorithms for encryption and authentication. Recently, modes that combine encryption and authentication together are being proposed. This feature is especially beneficial in case of hardware implementations, as it allows for a substantial decrease in the circuit area and power compared to traditional schemes.

In this thesis, we first characterize candidates of the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR). Then we discuss light-weight candidates from the round 1 submissions namely ACORN, SILC (SImple Lightweight CFB) and Joltik. We first implement the full width designs of these candidates targeting Xilinx Spartan-6 and Artix-7 FPGAs. Later, we optimize these designs for low-area applications. Lastly, we compare the results of the implementations with other published results.

Chapter 1: Introduction

In this chapter we first discuss the basic security objectives of any cryptographic system and then the methods that assure the security objectives. Later, we introduce the topic of Authenticated Encryption and it's advantages over traditional schemes.

1.1 Security Services of Cryptography

There are four security objectives of cryptography that form the structure of security services, namely

1. Confidentiality.
2. Data Integrity.
3. Authentication.
4. Non-repudiation of Message.

1.1.1 Confidentiality

This is a service that protects the data from unauthorized disclosure. In simpler words, confidentiality is said to be ensured if and only if the sender and receiver of the message can get access to it. Figure 1.1 shows the confidentiality service. Confidentiality ensures that Eve, who is not a part of the communication and is unauthorized will not get access to the message that is sent by Alice to Bob.

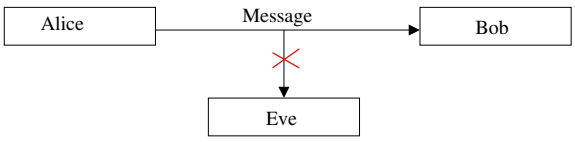


Figure 1.1: Confidentiality

1.1.2 Data Integrity

Data integrity can be defined as an assurance that the data received is unaltered or unchanged during transmission. Data modifications include insertion of bits, deletion of bits or substitutions. Figure 1.2 shows the integrity service. Integrity ensures that Bob detects the modification of data done by Eve, who is an unauthorized person.

Figure 1.2: Integrity

1.1.3 Authentication

Authentication is a process of verifying the identity of a user who wishes to access the information, this type of authentication is called as peer-entity authentication. Another type of authentication is data origin authentication which ensures that the data is originally coming from the actual sender and not from any third party. Figure 1.3 shows the authentication service. Authentication guarantees for Bob that the message is originated from Alice who claims to be the author.

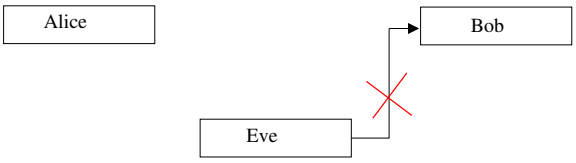


Figure 1.3: Authentication

1.1.4 Non-repudiation of Message

Non-repudiation of a message is a service which prevents both communicating parties from denying sending or receiving the message. Digital signature technique is one way assuring Non-repudiation

of a message.

In practice we often need both confidentiality and authentication for example, medical information sent my doctors has to be both confidential and authentic. Traditionally these two services are achieved using separate algorithms. Confidentiality is provided by encryption, various confidentiality-only modes of operations are explained in section1.2. Authentication can be provided by message authentication codes (MACs). These authentication techniques are covered in section 1.3

1.2 Confidentiality-only Modes of Operations of Block Ciphers

A mode of operation of a block cipher is an algorithm that uses symmetric key block cipher algorithm to provide a cryptographic service such as confidentiality or authentication. This section covers the basic modes of operation provide only confidentiality

1.2.1 Electronic Code Book (ECB) Mode

ECB is the most basic mode of operation. The input message is broken into blocks of length equal to the block size of the cipher. Each block is encrypted separately. In the same way at the receiver side the ciphertext is broken into blocks with block size length and each block of ciphertext is decrypted separately. Figures 1.4, 1.5 explain the operation of ECB encryption and Decryption respectively. The disadvantage of ECB mode is that it produces identical ciphertext blocks for identical input blocks and that the block cipher needs encryption and decryption modes.

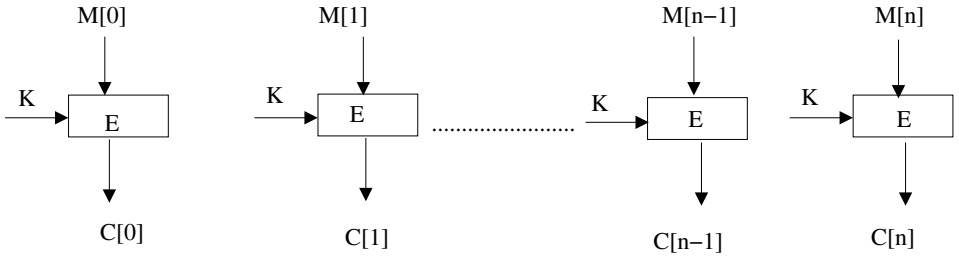


Figure 1.4: ECB Encryption

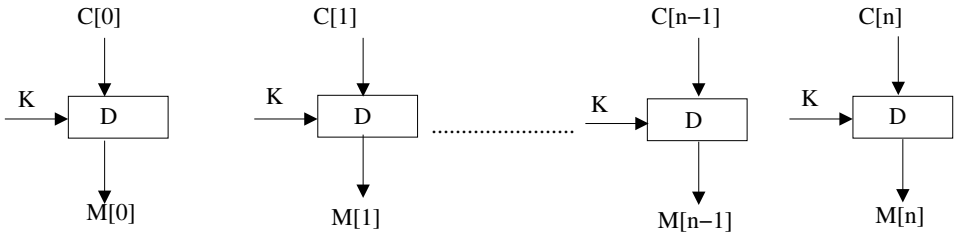


Figure 1.5: ECB Decryption

1.2.2 Cipher Block Chaining(CBC) Mode

Each subsequent plaintext block is XORed with the previous ciphertext block. In order to make each message unique the first block of plaintext is XORed with an Initilization Vector(IV). At the receiver side, the plain text is obtained by decrypting the ciphertext and XORing it with IV for the first block of ciphertext and with previous plaintext for all subsequent ciphertexts. The main drawback of CBC is that it cannot be parallelized and needs a block cipher that can encrypt and decrypt. Figures 1.6, 1.7 explain the operation of CBC encryption and decryption respectively.

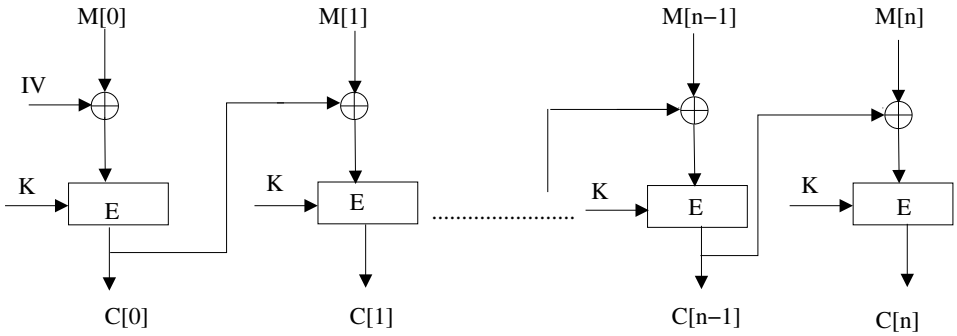


Figure 1.6: CBC Encryption

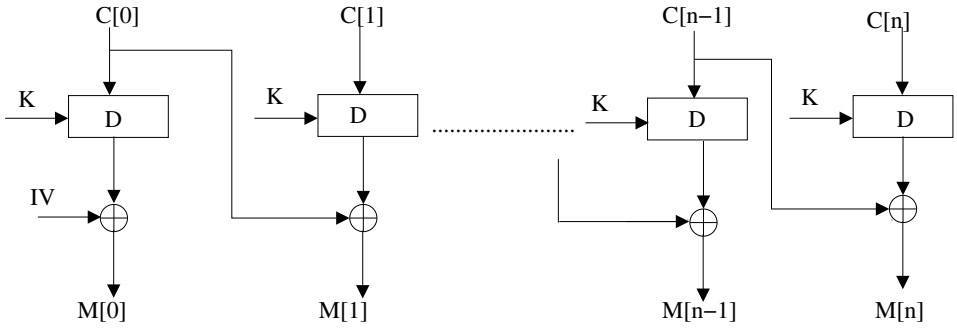


Figure 1.7: CBC Decryption

1.2.3 Cipher Feedback (CFB) Mode

In CipherFeedback mode(CFB) the first block of ciphertext is obtained by encrypting an Initilization Vector (IV) and XORing it with a plaintext block. The resulting ciphertext block is supplied as the feedback for generating next block of ciphertext. CFB mode is inverse free which means that,an encryption algorithm alone is sufficient for both encryption and decryption. The advantage of CFB is that it doesn't need any padding. CFB is self-synchronizing which means that loss in a part of ciphertext will only affect a part of plaintext. Figures 1.8, 1.9 explain the operation of CFB encryption and decryption respectively.

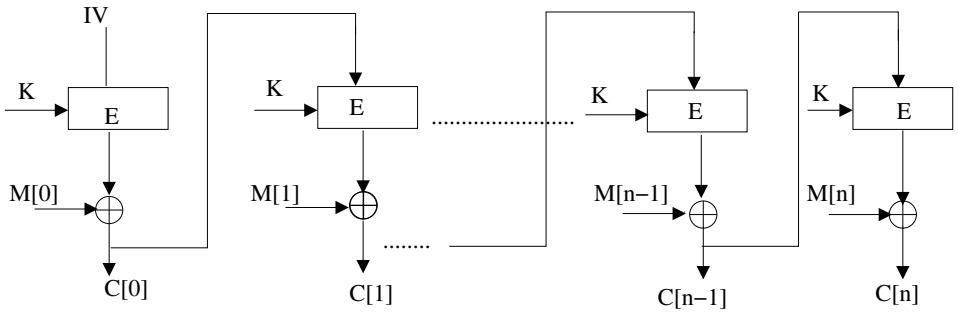


Figure 1.8: CFB Encryption

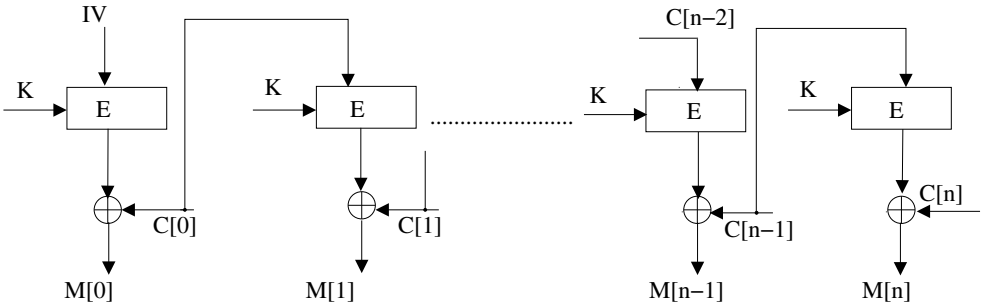


Figure 1.9: CFB Decryption

1.2.4 Output Feedback(OFB) Mode

Operation of OFB is similar to that of CFB, however unlike CFB the feedback in OFB is the output of the encryption function. In OFB, every operation depends on previous operations there by it cannnot be parallized. OFB is inverse free, meaning both encryption and decryption operations can be performed using encryption algorithm alone. Figures 1.10,1.11 explain the operation OFB encryption and decryption respectively.

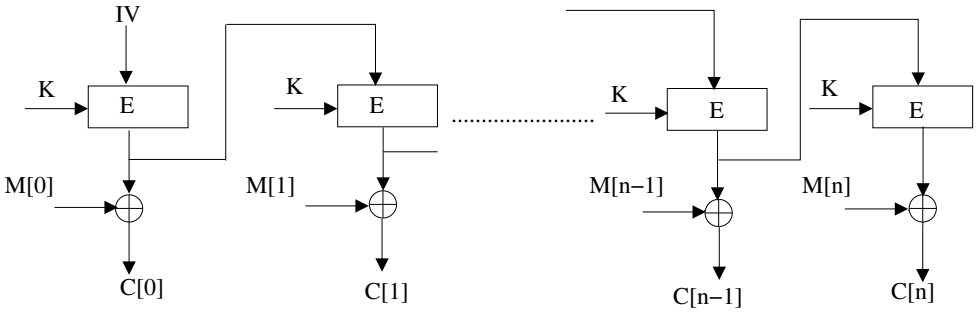


Figure 1.10: OFB Encryption

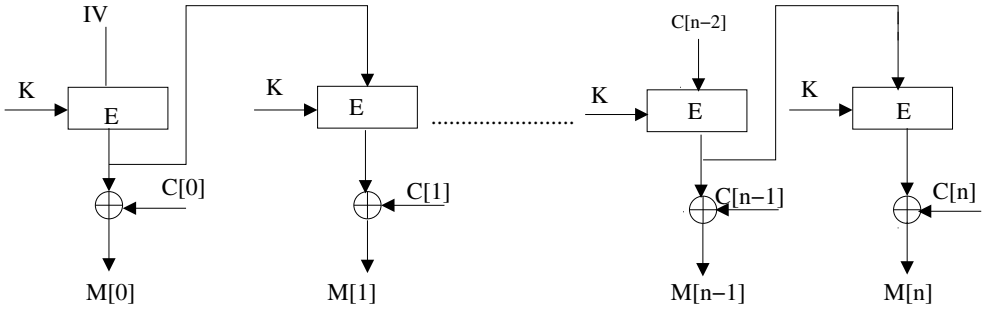


Figure 1.11: OFB Decryption

1.2.5 Counter (CTR) Mode

The inputs to an encryption function in CTR mode are a plaintext(M), key(K) and a counter (ctr) where ctr is an n-bit string. First, the ctr is encrypted and then the plaintext is XORed with the encrypted value to get the first block of ciphertext (C). Then ctr is incremented for generating the subsequent ciphertext blocks. The decryption is similar to encryption with M replaced by C. Figures 1.12,1.13 explain the operation of CTR encryption and decryption respectively.

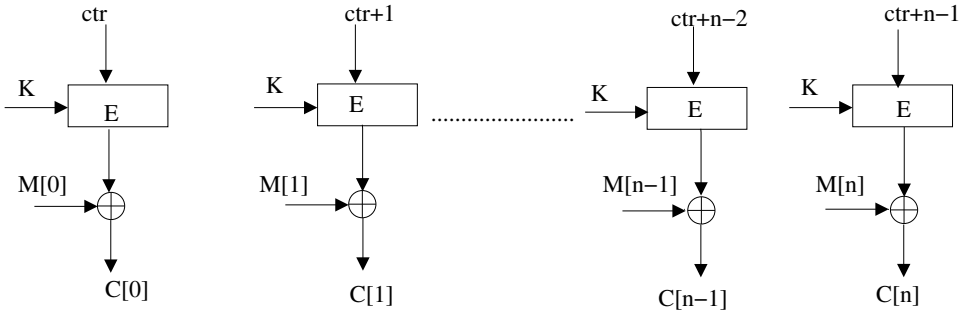


Figure 1.12: CTR Encryption

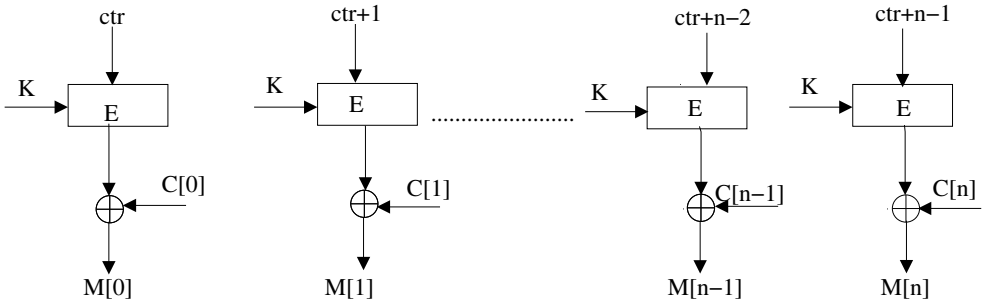


Figure 1.13: CTR Decryption

1.3 Authentication Techniques

As explained earlier in 1.1.3, the purpose authentication is to ensure that the data comes from the person who claims to be the sender. This section covers a detailed explanation of authentication techniques.

1.3.1 Cryptographic Hash Functions

A hash function is basically a transformation that takes a variable size input and returns it's hash value, which is a fixed length string. Hash functions are used as components for MACs. The input to any hash function is called as message and the output is called as message digest. The length of the message is arbitrary but the length of the output is fixed. Figure 1.14 explain the operation of a hash function.

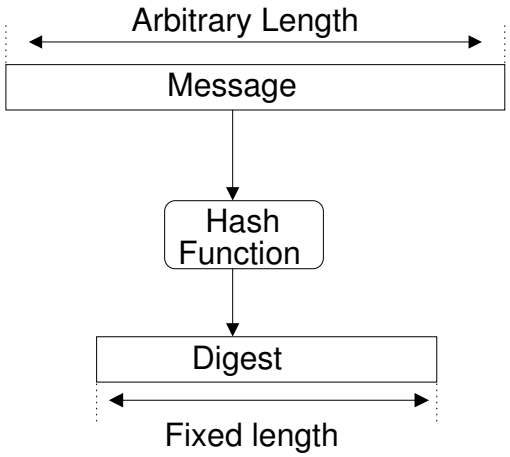


Figure 1.14: Operation of a Hash Function

The basic requirements of a hash function are as listed below.

- It is a one-way function.
- It must be easily computable.
- The output must be of fixed length.

1.3.2 Message Authentication Code(MAC)

MAC is basically a function which takes a secret key and a message of arbitrary length as input and gives out a unique fixed length MAC as output. MACs help to assure the message's origin(authentication) and also detect any changes in the message(integrity). Operation of MAC is similar to that of a hash. Figure 1.15 explain the operation of a MAC algorithm

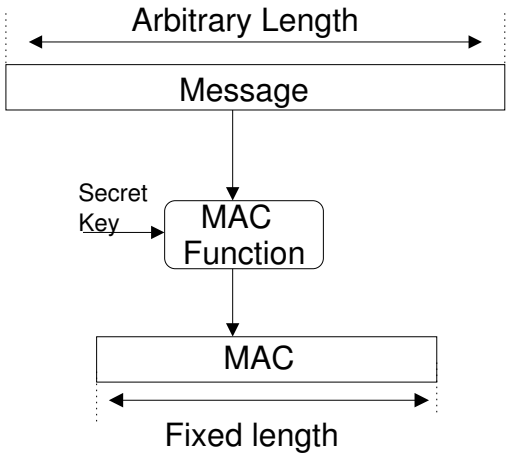


Figure 1.15: Operation of a MAC Function

The basic requirements of a MAC are as listed below

- It should contain a key.
- Fixed length output.
- It must be computationally easy.

The security requirement of a MAC algorithm is that it must be computationally infeasible to calculate m' and MAC(m') with m and MAC(m) given such that m', m are different.

Construction of MACs

A MAC can be built from a hash function or a block cipher. Based on the way they are built there are two types of MACs

1. Hash function based MACs.
2. Block cipher based MACs.

Hash Function Based MACs

A hash function based MAC is a function that is constructed with a combination of cryptographic hash function and a secret key (K). It takes a message (m) of arbitrary length and a secret key as inputs and return the fixed length MAC as output. Figure 1.16 explain the operation of a HMAC

function. Given a cryptographic hash function H, opad is the outer padding, ipad is the inner padding and | denotes concantenation then the definition of HMAC is

$$\text{HMAC(K,m)} = \text{H((K xor opad) | H ((K} \oplus \text{ipad)|m)).}$$

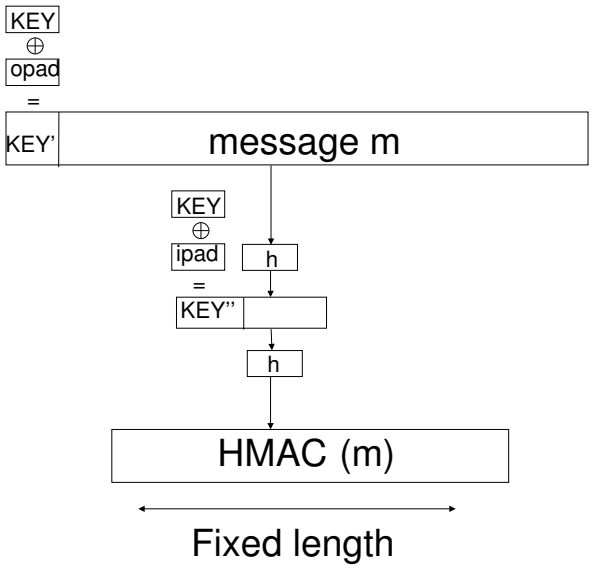


Figure 1.16: Operation of a HMAC Function

Goals of HMAC

The goals of HMAC are listed below:

- The underlying hash function must be easily replaced with the latest and secure hash function.
- Use the hash fucntions that are readily available without any modifications.

Block Cipher Based MACs

A message authentication code can also be built using a block cipher as the underlying primitive.

1. **Cipher Block Chaining Message Authentication Code (CBC-MAC)**

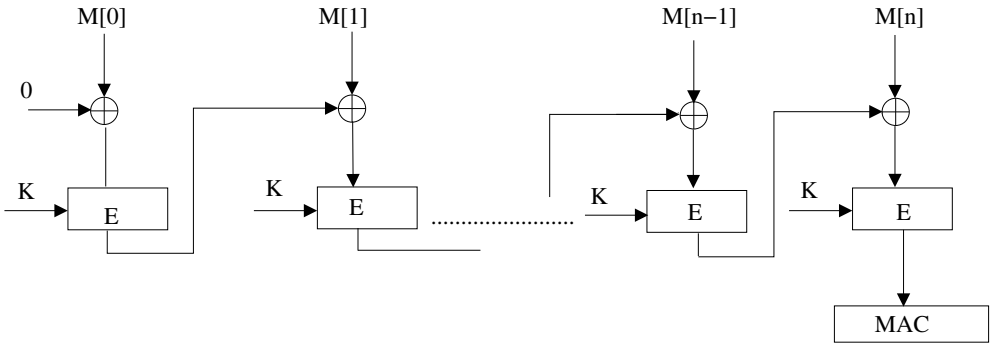


Figure 1.17: Operation of the CBC-MAC Algorithm

Operation of CBC-MAC

The MAC of the message is calculated by encrypting the message using a block cipher in CBC mode with an initialization vector of all zeros. The output of the first encryption is XORed with the next message block. This kind of structure ensures that a change in any bit of the plaintext will cause change in the final output. In CBC-MAC the length of the message has to be a multiple of n where n is the block size of the underlying block cipher. Figure 1.17 explain the operation of CBC-MAC.

2. **Parallelizable Message Authentication Code(PMAC)**

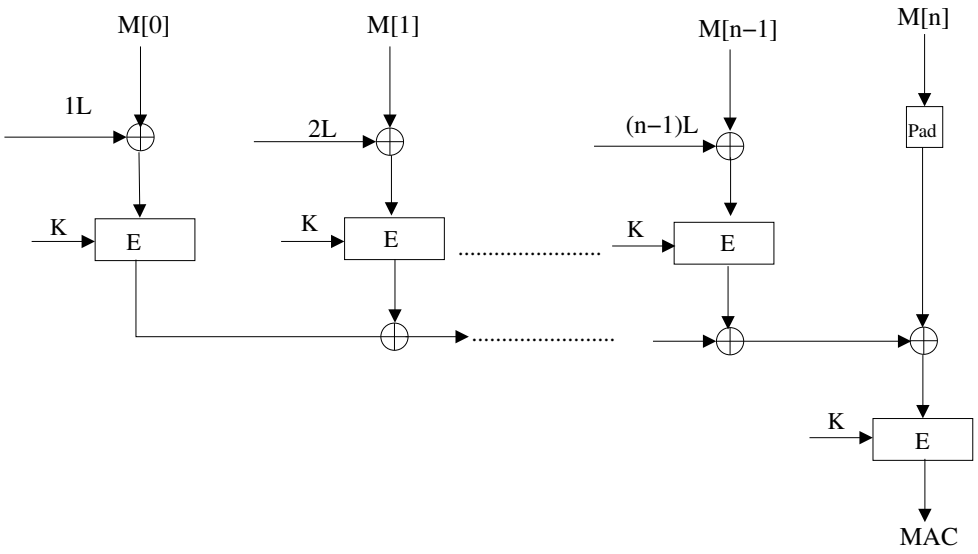


Figure 1.18: Operation of the PMAC Algorithm

Operation of PMAC

The inputs to the PMAC algorithm are a key (K) with k bits length, message (M) of length n bits. It uses three operations which are addition , multiplication and final. L is calculated by encrypting a string of zeros of length n and adding the output with a string of all zeros and least significant bit fixed to zero and final (L) is 1's compliment of L. Each block of message is added with iL for i $\geq$ 1. The operation of PMAC is illustrated in the figure 1.18

1.4 Authenticated Encryption

This section introduces the topic of Authenticated Encryption and focuses on it's advantages and different composition schemes.

1.4.1 What is Authenticated Encryption?

Authenticated encryption (AE) is primarily a combination of authentication and encryption that provides both privacy and authenticity of the data that is encapsulated. Any scheme that provides authenticated encryption takes the input plaintext (m), and key (K) and provides ciphertext (C) and

a tag (T) as output. Tag is considered as a checksum of the message and is used to check whether the correct ciphertext is received. Another class of AE schemes is authenticated encryption with associated data (AEAD) which supports both data that needs encryption along with authentication and data that only needs authentication. Figure 1.19 shows the basic block of an AEAD algorithm.

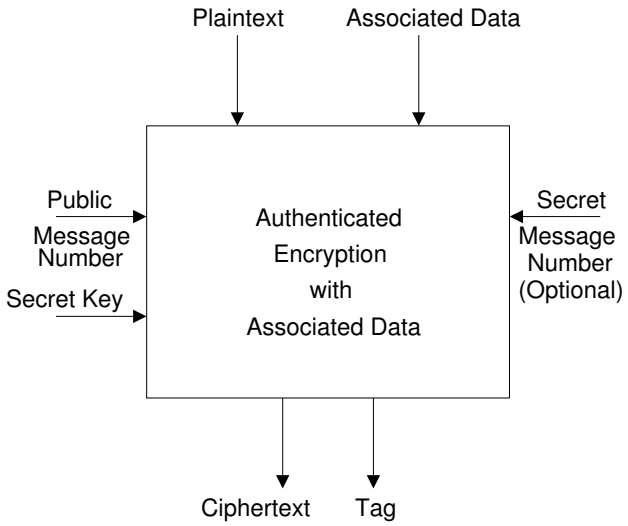


Figure 1.19: Authenticated Encryption with Associated Data

1.4.2 Advantages of Authenticated Encryption

Combining authentication and encryption into one single algorithm in hardware might possibly provide the advantages listed below.

- Area requirement could be smaller for a single algorithm there by reducing the cost.
- Designs with smaller area requirement consume less power there by it is a good solution for low-power applications.
- A combined algorithm needs only a single key and so has a slight advantage in the issues of key management and key storage.

1.4.3 Composition Schemes

Any AE scheme is basically a combination of an encryption algorithm and an authentication algorithm. There are three types of composition schemes for achieving authenticated encryption and they differ in the way these two algorithms are combined.

1. **Encrypt-then-MAC(EtM)**

   In this scheme a message is first encrypted and the tag is calculated by taking the MAC over the obtained ciphertext. And on the receiver's side first the tag gets verified and if it matches decryption will take place to get the plaintext. Figure 1.20 shows the operation of EtM composition scheme.

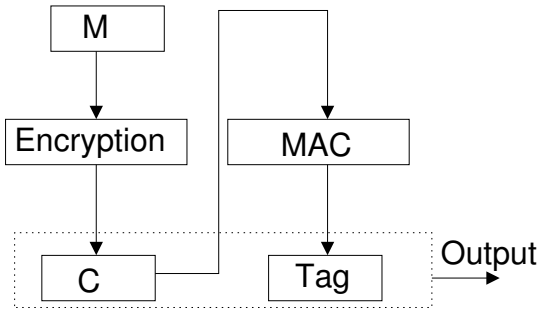


Figure 1.20: Operation of EtM

2. **MAC-then-Encrypt(MtE)**

   In this scheme first the tag is calculated by taking the MAC over the message. The obtained is tag is then appended to the message and tjhe resultant is ecncrypted to generate the ciphertext. And on the receiver's side first decryption will takes place to get plaintext and tag pair and then verifies the tag. Figure 1.21 shows the operation of MtE composition scheme.

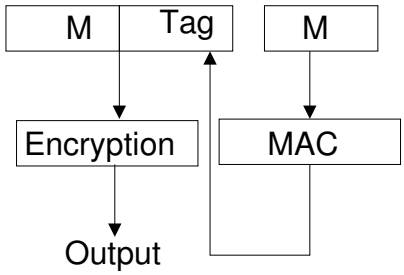


Figure 1.21: Operation of MtE

3. **Encrypt-and-MAC (E&M)**

   The message is encrypted to get the ciphertext and the tag is also calculated on the original message. And on the receiver's side first decryption takes place to get the plaintext and then verifies the tag. Figure 1.22 shows the operation of E&M composition scheme.

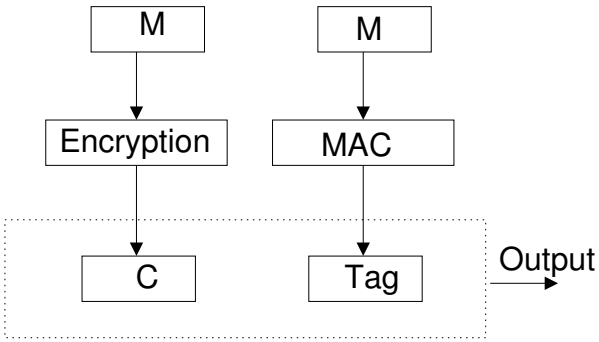


Figure 1.22: Operation of E&M

Chapter 2: Classification of the CAESAR Candidates

In this chapter we first discuss CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) and it's requirements. Later, we characterize all submissions under certain features.

2.1 Introduction

CAESAR has called for submissions of authenticated ciphers on 2/1/2014. The interest of the competition mainly lies in finding a cipher that has advantages over AES-GCM and is suitable for a very wide range of applications. The competition has certain requirements that every submission must comply with.

2.1.1 Functional Requirements of the CAESAR Competition

The requirements of CAESAR competition are listed [2]:

- The cipher must provide both integrity and confidentiality to Plaintext and Secret message number and also integrity to Associated data and Public message number i.e., the cipher must be Authenticated Encryption with Associated Data(AEAD), which is a special case of Authenticated Ciphers.
- Ciphers must not leak any information other than the length of plain text via the length of cipher text.
- The submission must clearly specify the recommended parameters. The number of recommendations must not exceed 10.
- It must be possible to recover the plaintext and the secret message number from the ciphertext, associated data, public message number, and key.

2.2 Design Classification

The Round 1 submissions include a variety of designs. The characteristics of the candidates are discussed in this section.

2.2.1 Type

An Authenticated Cipher can be based on Block Ciphers, Tweakable Block ciphers, Stream Ciphers or Permutations. Some submissions are a combination of two or more types.

Block Cipher

A block cipher is one of the methods of encrypting a plain text to produce the corresponding cipher text in which the key and algorithm are applied block wise rather than to single bit at a time. The block cipher takes key K and plaintext M as input and returns ciphertext C as the output. Figure 2.1 shows how a block cipher works.

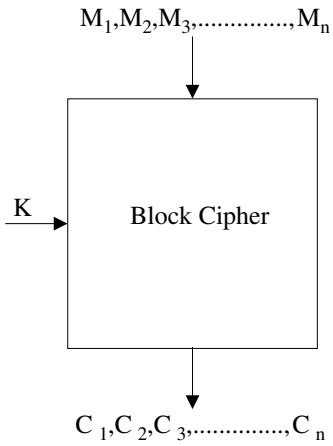


Figure 2.1: Block Cipher

Stream Cipher

In a stream cipher a sequence of plaintext digits m is encrypted into a sequence of ciphertext c digits. Unlike block ciphers, the key K and algorithm are applied bit wise. Figure 2.2 shows how a stream cipher works.

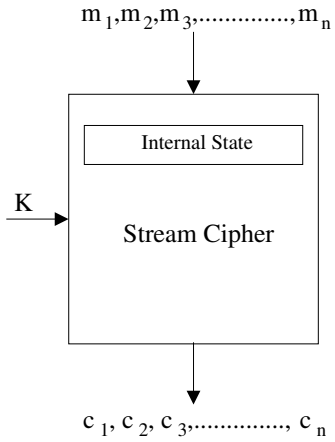


Figure 2.2: Stream Cipher

Permutation Based

Few candidates of CAESAR uses a fixed length permutation as their underlying structure.

Tweakable Block Cipher

Tweak is formed by concatenating public parameters, block counters and some cipher dependent parameters, for example a string of bits. Tweak is often used to generate tweakey (formed by concatenating tweak with key), which is used for encryption. Figure 2.3 shows how a tweakable block cipher works.

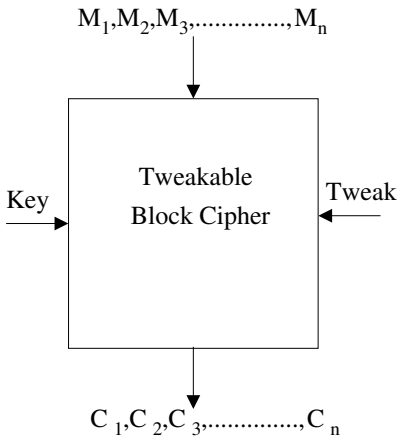


Figure 2.3: Tweakable Block Cipher

2.2.2 Features

1. An AE scheme can be called as *parallelizable* if they are not sequential and can offer a possibility for parallelizing the computations between distinct block cipher calls. In an AE scheme either the encryption, the decryption or both can be parallelizable.

   - **Parallelizable Encryption** In any AE scheme, if processing of the ith plaintext block doesn't depend on jth plaintext block for i$\neq$ j then the encryption of that particular AE scheme is parallelizable.

   - **Parallelizable Decryption** In any AE scheme, if processing of the i th cipher text block doesn't depend on jth cipher text block for i $\neq$ j then the decryption of that particular AE scheme is parallelizable.

2. **Online**: A cipher is called online if the encryption of i-th input block M_i depends only on the previous blocks $M_1, \ldots, M_{i-1}$ [4].

3. **Inverse free**: If an AE scheme only requires either encryption or decryption then it can be called as an Inverse free AE scheme. Inverse free schemes requires low memory and area.

4. **Masking**: Some of the block cipher based AE schemes mask the inputs and output for additional security. There are three types of masking methods that are adopted by the AE schemes.

   - AX: Addition and XOR arbitrary input.
   - Doubling: Inputs and outputs are XORed with a variable which is key dependent and is incremented by doubling in Galois Field.
   - Galois-Field Multiplication (GFM): Inputs and outputs are multiplied in Galois-Field with a variable that depends on the key.

5. **Passes**: The AE schemes can be classified into one pass and two pass depending on the way they treat the data:

One-pass: One pass mode executes encryption and authentication at the same time.

Two-pass: Two-pass mode executes one algorithm first and then executes another.

6. **Tag Verification type**: There are two options to verify a tag, it can be done before the decryption or after the decryption is done. The verification type depends on the composition scheme they follow and there are three different composition schemes

   - **Encrypt-then-MAC (EtM)** As discussed in 1 the tag is verified before the decryption if the tag doesn't match, the decryption will not take place.
   - **MAC-then-Encrypt (MtE)** and **Encrypt-and-MAC (EM)** As discussed in 2 3 the tag is verified after the decryption is done.

Table 2.1 shows the classification of CAESAR Round 2 candidates and table 2.2 shows the Round1 candidates that did not advance to Round 2.

Table 2.1: Classification of CAESAR Round Two candidates

Candidate	Type	Primitive	Modified	Masking	Parameters					Features						
					Tag Size	Key Size	Block/State Size	PMN Size	Word Size	Max AD/M size	Rounds	Parallelizable E/D	Mode of operation	Online E/D	Inverse-free	Passes	Verification
ACORN v1v2[50]	SC	LFSR	bit based FSR		128	128	1/293	128		$<2^{64}/<2^{64}$	10	•/•	–	–/–	•	2	MtE
AEGIS v1[24]	BC	AES	last round not used		128	128	128	128	128	$\leq 2^{64}/\leq 2^{64}$	9	•/•	–	–/–	•	2	MtE
AES-COPA v1v2[7]	BC	AES		D	64|32|16	128	128	128		V/V		•/•	–	•/•	•	2	MtE
AES-JAMBU v1v2[25]	BC	AES,JAMBU			64	128	128	64		$<2^{64}/<2^{64}$		–/–	–	–/–	•	2	MtE
AES-OTR v1v2[39]	BC	AES		D	96	128	128	96		$\leq 2^{64}/\leq 2^{64}$		○/○	–	•/•	•	1	EM
AEZv1v3v4[22]	BC	AES	key scheduling		128	128	128	96		V/V		•/•	–	–/–	•	2	MtE
ASCON v1v1.1 [17]	P	Monkey Duplex	own permutation		128	128	64/320	128	64	$<2^{64}/<2^{64}$	20	–/–	–	•/•	•	2	MtE
CLOC v1v2 [27]	BC	AES			64	128	128	96	8	V/V		–/–	•	•/•	•	2	EtM
CLOC v1v2 [27]	BC	TWINE			32	80	64	48	8	V/V		–/–	•	•/•	•	2	EtM
Deoxys v1v1.3 [30]	TBC	Deoxys-BC	Round tweakey operation		128	128|256	128	64	128	V/V	14	•/•	–	–/–	–	2	MtE
ELmD v1 v2.0 [16]	BC	AES	Encrypt-mix-Encrypt	D	128	128	128	64		$\leq 2^{64}/\leq 2^{64}$	12	•/•	–	•/•	•	2	MtE
HS1-SIV v1v2 [35]	HS1	PRF			128	256	256	96	32	V/V	20	•/•	•	–/–	•	2	MtE
ICEPOLE v1v2 [40]	P	Sponge	own permutation		128	128	1280	128	64		12	•/•	–	•/•	•	2	MtE
Joltik v1v1.3 [31]	TBC	Joltik-BC			64	128	64	32	4	V/V	24	•/•	–	•/•	–	2	MtE
Ketje JR v1 [9]	P	KECCAK-p[200]	variable tag length		128	$\leq$182	16/200	128		V/V	12	–/–	–	•/•	•	2	MtE
Ketje SR v1 [9]	P	KECCAK-p[400]	variable tag length		96	$\leq$382	32/400	80		V/V	12	–/–	–	•/•	•	2	MtE
Keyak v1 v2 [10]	P	KECCAK-p[800,1600]	key stream generation		128	$\leq$2031	544/800	128	8	V/V		•/•	–	•/•	•	2	MtE
Minalpher v1 v1.1 [45]	TBC	TEM			128	128	/256	104	4	$<2^{104}$-1/$<2^{104}$-1	17	•/•	•	–/–	–	2	EtM
MORUS v1 v1.1 [51]		LRX			128	128|256	128/640|1280	128	32	$<2^{64}/<2^{64}$	10	–/–	–	•/•	•	2	MtE
NORX v1v2 [33]	P	Monkey Duplex,RX	padding		128|256	128|256	128/512|1024	64|128	32|64	V/V	$\leq$63	○/○	–	•/•	•	2	MtE
OCB v1 [36]	BC	AES	hashing	D	96|128	192|256	128	128	128	V/V		•/•	–	•/•	–	2	MtE
OMD v1.0 v2.0 [15]	CF	SHA-256,SHA-512			32-256	128-256	256	96-256	32	V/V		–/–	•	•/•	•	2	MtE
PAEQ v1 [11]	P	PPAE,AESQ Permutation			160|512	128|160	368/128	128|96	32	V/V	20	•/•	–	–/–	•	2	MtE
Π-Cipher v1 v2 v2.0 [18]	P	triplex			256|512	128|256	512/256	32|128	16|32|64	$<2^{64}$-1/$<2^{64}$-1	4	•/•	–	•/•	•	2	MtE
POET v1 v2.0 [3]	BC	ECB	applies e-AXU function		128	128	128	128		V/V	18	•/•	–	•/•	•	1	EM
PRIMATES v1 v1.02 [6]	P	PRIMATE	own permutation		160|240	80|120	40/200|280	120|160|240		V/V	6|12	–/–	–	•/•	•	1	EM
SCREAM v1 v3 [19]	TBC	TAE	key scheduling		128	128	128	96	16	V/V		•/•	–	•/•	•	2	MtE
SHELL v1v2.0 [49]	BC	AES,AES[4] permutations		D	128	128	128	64|80		$<2^{63}/<2^{63}$	10	•/•	•	•/•	–	2	MtE
SILC v1 v2 [28]	BC	AES			64	128,80	128	96	16	$<2^{64}$-1/$<2^{64}$-1	10	–	•	•/•	•	2	EtM
SILC v1v2 [28]	BC	PRESENT			64	80	128	48	16	$<2^{64}$-1/$<2^{64}$-1	10	–	•	•/•	•	2	EtM
SILC v1 v2 [28]	BC	LED			64	80	128	48	16	$<2^{64}$-1/$<2^{64}$-1	10	–	•	•/•	•	2	EtM
STRIBOB v1 v2 [44]	P	Sponge	own permutation		128	192	256	128	8	V/V	12	•/•	–	•/•	•	2	MtE
Tiaoxin v1 v2 [41]	BC	AES			128	128	128	128	128	$<2^{128}$-1/$<2^{128}$-1	35	•/•	–	•/•	•	2	MtE
Trivia-ck v1v2 [14]	SC	Trivia-SC	key generation		128	128	64/384	64	32	V/V		–	–	–/–	•	2	MtE

Table 2.2: Classification of CAESAR Round One Candidates

Candidate	Type	Primitive	Modified	Parameters								Features				
				Masking	Tag Size	Key Size	Block/State Size	PMN Size	word Size	Max AD/M size	Rounds	Parallelizable E/D	Mode of operation	Online E/D	Inverse/free	Passes	Verification
++AE v1[43]	BC	IOBC,IOC	uses bit stealing	AX	128	128	128	64	64	$<2^{64}/<2^{64}$		•/•	–	–/–	–	2	MtE
AES-CMCC v1.1[47]	BC	CBC	CMAC padding		128	128	128	32|16		$<2^{64}/<2^{64}$	10	•/•	•	–/–	–	2	EtM
AES-CPFB v1[38]	BC	AES			128	128	128	96		$<2^{35}$-1/$<2^{67}$-1	10	•/–	•	•/•	•	2	MtE
Artemia v1 [29]	P	JHAE	padding		128|256	128|256	128|256	128		V/V		–/–	–	•/•	•	1	EM
AVALANCHE v1 [5]	BC	AES,PCMAC	key scheduling		128	448	128	128		$\leq 128(2^{48}$-1)/NC		•/•	–	•/•	•	1	EM
CBA v1.1[26]	BC	AES		D	64|32|96	128	128	96		$\leq 2^{64}/\leq 2^{64}$		•/•	•	•/•	•	1	EM
Enchilada-128 v1.1[21]	SC,BC	ChaCha,Rijndael	whitened Rijndael		128	256	128|256	64		$\leq 2^{64}/\leq 2^{64}$	12,10	•/•	–	•/•	•	2	MtE
Enchilada-256 v1.1[21]	SC,BC	ChaCha, Rijndael	whitened Rijndael		128	256	256	64		$\leq 2^{64}/\leq 2^{64}$	20,14	•/•	•	•/•	•	2	MtE
ifeed[AES] v1 [55]	BC	AES	PMAC for tag generation	D	128	128	128	96	64			•/–	•	•/•	•	2	MtE
Julius v1 [8]	BC	AES CTR|ECB	pre computations	G	128	128	128	96		$\leq 2^{64}$-1/$\leq 2^{64}$-1		•/•	•	–/–	• | –	2	MtE
KIASU v1 [32]	TBC	KIASU-BC	10^*padding		128	128	128	32		V/V	11	•/•	–	•/•	–	2	MtE
LAC v1 [54]	BC	AES, LBlock-s	no reuse of round keys		64	80	1/144	64		V/V		–/–	–	•/•	–	1	EM
PROST v1.1 [34]	P	PROST			128|256	128|256	64|128/256|512	64|128		V/V	16|18	•/•	–	•/•	•	2	MtE
Raviyoyla v1 [48]	SC	MAG v2,random function			128	256	128	128		V/V		–/–	–	–/–	•	2	MtE
Sablier v1 [53]	SC	LFSR	internal state		32	80	208	80	16	$<2^{19}/<2^{19}$	64	–/–	–	–/–	•	2	MtE
Silver v1 [42]	BC	AES	key scheduling		128	128	128	128	128	$<2^{50}$-1/$<2^{50}$-1	10	•/•	–	•/•	–	2	MtE
Wheesht v1 [37]	SC	modular addition,rotation,XOR	No S-Box		256	512	256/512	128		V/V		•/•	–	–/–	•	2	MtE
YAES v1 v2 [12]	BC	AES	Uses tweaked CTR		128	128	128	127		$\leq 2^{51}$	10	•/•	–	•/•	•	2	MtE

Legend

- Type
  - BC: Block Cipher
  - SC: Stream Cipher
  - P: Permutation
  - CF: Compression function
  - TBC: Tweakable block cipher
- Primitve
  - LRX: Logical operations, Rotation and XOR
  - AX: Addition and XOR.
- Masking
  - D: Doubling
  - AX: Addition and XOR.
  - GFM: Galois-Field Multiplication
- Features
  - NC: No Constraint.
  - • Yes
  - –: No
  - •/•: Fully/Fully
  - V/V: Variable/Variable
  - ○/ ○: Partly/Partly
  - –/–: No/No
- Verification Type
  - MtE: MAC then Encrypt
  - EtM: Encrypt then MAC
  - EM: Encrypt and MAC

Chapter 3: Design Decisions

3.1 Candidate Selection

From all the candidates submitted at CAESAR we have chosen SILC, ACORN and Joltik as all the 3 candidates are lightweight. These 3 candidates are a variety of design in the sense that they are based on different kinds of cipher. SILC is block cipher based, Joltik is tweakable block cipher based and ACORN is stream cipher based. So, implementing these 3 candidates will cover almost all the design types submitted at CAESAR.

3.2 Hardware Interface for Fullwidth Designs

The top level interface for all the candidates is the George Mason's Hardware API for Authenticated Ciphers [23]. The authenticated cipher interface has pre and post processors in it. These processors receive and format the data into segments that can be used by ciphers, output. Using the interface the cipher core designed for every candidate is wrapped with a wrapper which has Serial-input/parallel-output (SIPO) and parallel-input/serial-output (PISO)in them,which allow fewer IO connections like 32 and 64 bits for public and secret data interfaces, which expands the available FPGA platforms on which designs can be implemented. The authenticated cipher interface provides below mentioned services

- 10* padding.
- Delimeters that provide information about end-of-input and end-of-text.
- Validates the decryption and verification of tag by buffering all the output until the "msg_auth_valid" signal is generated by the cryptographic core.

Figure 3.1 shows the top level hardware interface used for full width designs.

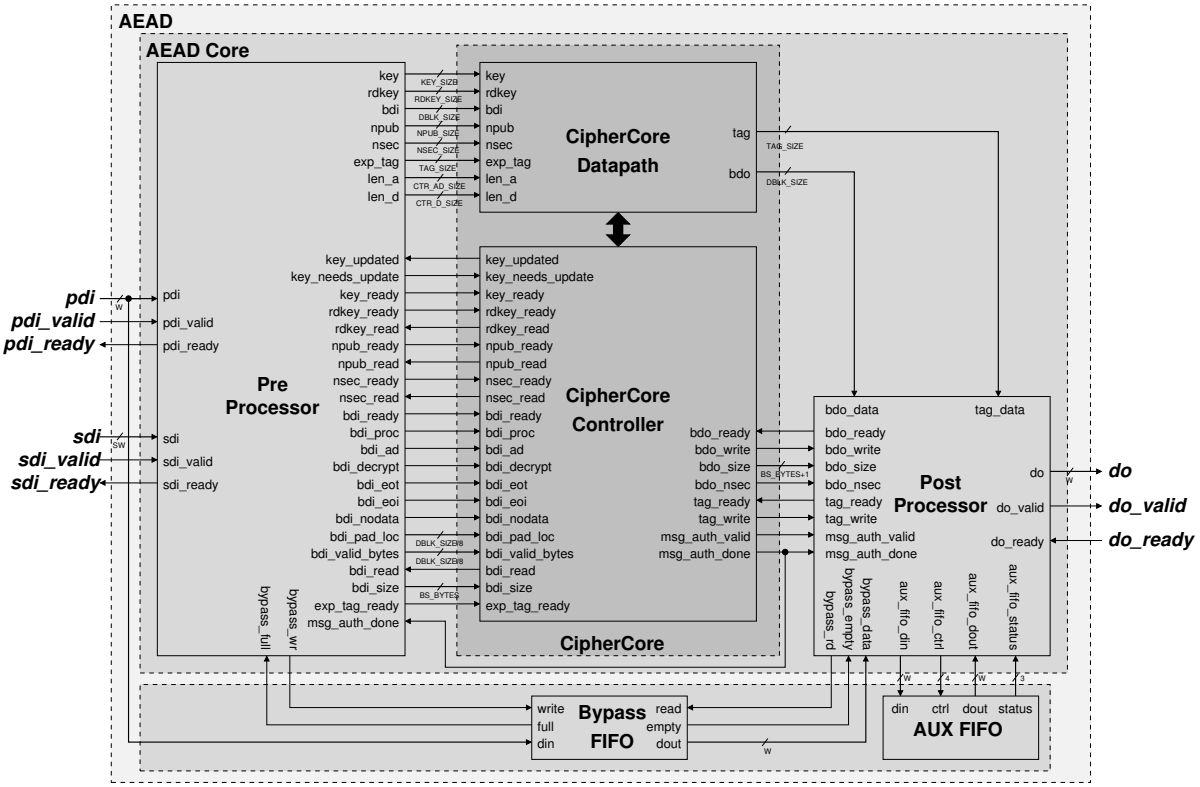


Figure 3.1: Hardware Interface for Fullwidth Designs

3.3 Lightweight Interface

While optimizing the designs for lightweight applications we have used the 16-bit interface developed by Pansayya Yalla of George Mason University. The interface has a common public data input (**pdi**) port for associated data, nonce, plaintext, ciphertext and tag. An additional signal *pdi_type* is used to specify the type of data that is coming on the bus. The key and secret message number come through the secret data input port. The data output port **do** carries the cipher text and tag. The interface consists of the following modules:

- Pre-processor which takes the public data in and sends the corresponding signals to the cipher core.
- A secret processor which takes the secret data in and passes the data to the cipher core. It also passes the error code signals to the post processor.

- Finally a post processor, which takes the signals coming from the cipher core and sends out the data.

Figure 3.2 shows the top level hardware interface used for light-weight designs.

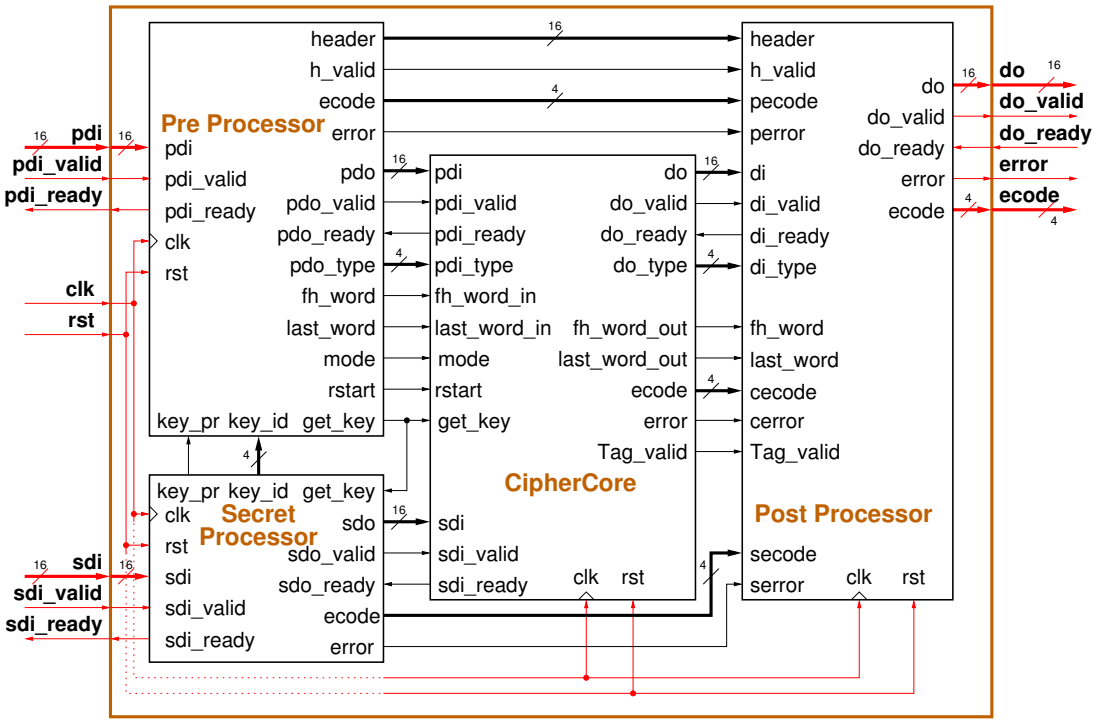


Figure 3.2: Hardware Interface for Lightweight Designs

3.4 Design Methodology

The implementations of all three candidates are developed using the specifications submitted to CAESAR Round 2. The designs were written, synthesized, and implemented using VHDL and Xilinx Webpack 14.7 ISE.

3.5 Functional Verification

The test vectors of all candidates are generated from the reference software implementations in C using Microsoft Visual Studio 2013. The reference software implementations of the CAESAR

candidates are obtained by downloading the entire SUPERCOP package. The test vectors generated are given as inputs to the testbenches and outputs are verified to validate the functionality of all the candidates. All the designs are verified using post place and route simulation.

3.6 Results Generation

The timing and resource utilization results are obtained by using Automated Tool For Hardware Evaluation (ATHENA). The target families of Xilinx FPGA are Spartan6, Artix7 and Virtex6. We have chosen these families as Spartan6 and Artix7 consume low power and are good to implement the lightweight designs where as Virtex6 is highspeed and so good choice for fullwidth implementation comparisons.

Chapter 4: SILC: SImple Lightweight CFB

In this chapter we present Our implementation of SILC. First, we will discuss the algorithm and in the later section we discuss the hardware implementations.

4.1 Introduction

SILC (SImple Lightweight CFB (CipherFeedBack)) is a mode of operation with a block cipher as the underlying base function. It is a lightweight function i.e., the hardware implementation cost is very low. It is suitable for use in constrained hardware devices.

4.1.1 Features

SILC doesn't need much precomputation other than key scheduling so less hardware is needed there by reducing computational cost. It also has low memory cost because it can work with two state blocks [28]. The encryption and decryption operations of SILC can be done with the use of the encryption function alone. Both encryption and decryption are online operations that means i-th input block M_i depends only on the blocks $M_1,\ldots,M_{i-1}$. It is inverse free which means it only requires either encryption for both encryption and decryption operations. It is a two pass scheme i.e., it executes authentication first and encryption later. It follows EtM composition scheme for verification of the tag which means it verifies the tag before decryption.

4.1.2 Recommended Parameter Set

SILC takes the following three parameters

1. Block cipher (E).
2. Length of the nonce (l_N).
3. Length of the tag (τ).

Table 4.1 shows parameters recommended by designers of SILC and the respective *param* value.

Table 4.1: Recommended Parameter Set of SILC

Parameter set	Block Cipher(E)	Length of the nonce (l_N)	tag length (τ)	param
aes128n12silcv1	AES-128	96	64	0xc0
aes128n8silcv1	AES-128	64	64	0xd0
present80n6silcv1	PRESENT-80	48	32	0xc4
led80n6silcv1	LED-80	48	32	0xc8

4.2 Encryption and Decryption

SILC uses fOur subroutines to perform the encryption and decryption operations:

1. HASH.
2. Encryption (ENC)
3. Psuedo Random Function (PRF)
4. Decryption (DEC)

These subroutines are covered in subsection 4.2.2. The subroutines use functions in their algorithm which are covered in subsection 4.2.1.

4.2.1 Functions Used in SILC

Length adjusting functions "zap" (zero appending function) and "zpp"(zero prepending function) are used to adjust the length of an input string to a non-negative multiple of n (n in the case of AES is 128) bits. The bit fixing function "fix1" is used to set the most significant bit of the input to one. The tweak function "g(X)" is defined as left shift with the rightmost output byte being the XOR of the leftmost two input bytes. The Length encoding function "Len(X)" is the standard encoding of the byte length of the string into bits of length n. The "msb" function is used to output the most-significant bits of the input function.

4.2.2 Subroutines Used in SILC

HASH Function

The HASH function takes key (K), nonce (N) concatenated with *param*, associated data (A) as the inputs and returns intermediate tag (V) as the output. It encrypts the zero prepended nonce in the case of an empty associated data string. In the case of a non empty associated data it encrypts the XORed value of associated data with the previous encrypted value of associated data. The output of the encryption function is then XORed with the length of the associated data and then sent to the tweak function (g). It returns intermediate tag(V) as the output. Figure 4.1 explains the operation of the HASH function. If the length of the associated data is zero then the part in the dotted box is not executed.

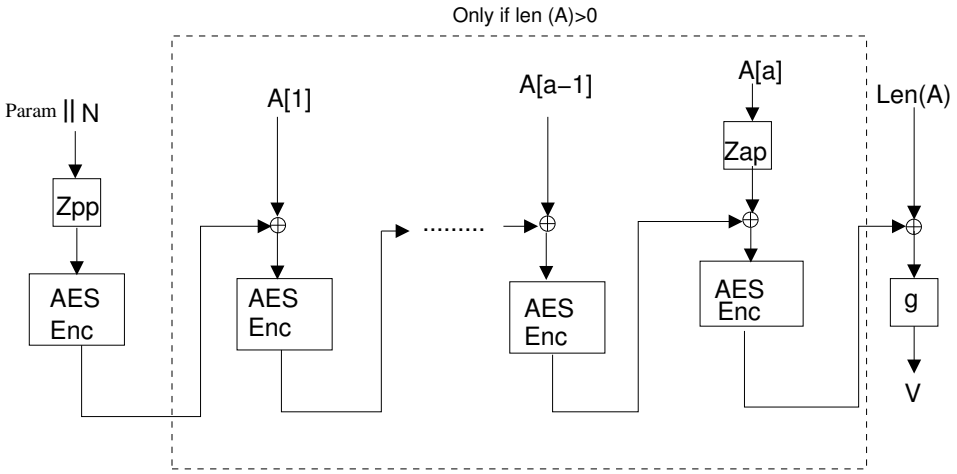


Figure 4.1: HASH Function

PRF Function

The PRF function takes the intermediate tag (V) and ciphertext (C) as inputs and returns tag (T) as output. First, the tweaked V is encrypted and then XORed with first block of the ciphertext. The outputs are continuously sent as feedback to the next block encryption. The tag is generated by taking the most significant 64 bits out of the tweak of last block's encryption. In the case of an empty ciphertext, the tag is generated by taking the most significant 64 bits of the output of encryption of tweaked V. Figure 4.2 explain the operation of the PRF function. If the length of the

ciphertext is zero then the part in the dotted box is not executed.

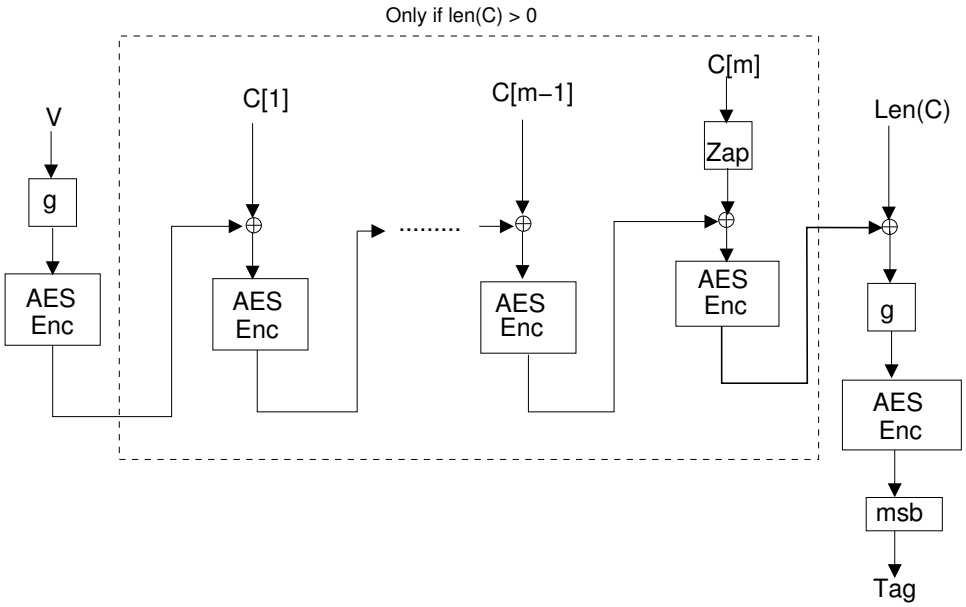


Figure 4.2: PRF Function

ENC Function

The ENC function takes the intermediate tag (V) and message (M) as inputs and returns ciphertext (C) as output. If the length of the message is zero then the cipher text is a string of zeros. In the case of a non-empty message, the first block of the cipher text is obtained by XORing the first block of the message with encrypted output of the intermediate tag (V). To generate the i^{th} cipher text block the i^{th} message block is XORed with encrypted value of the $(i-1)^{th}$ ciphertext by fixing it's most significant bit to 1. Figure 4.3 explains the operation of the ENC function.

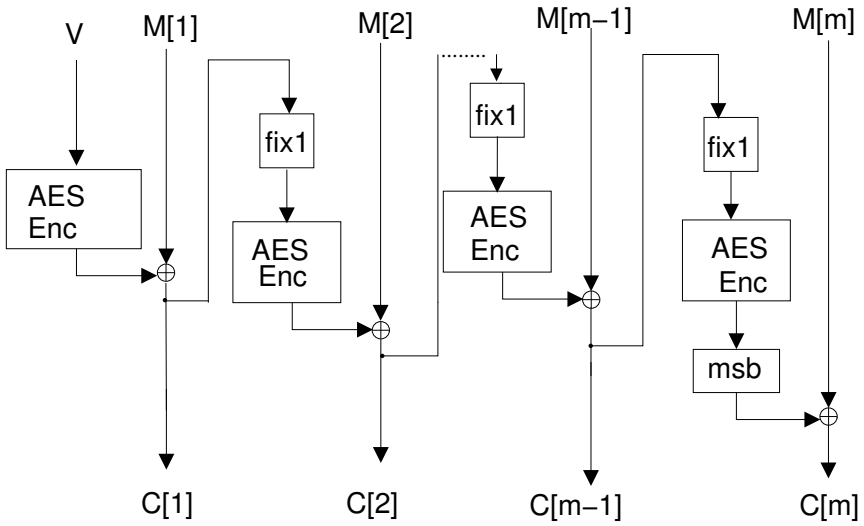


Figure 4.3: ENC Function

DEC Function

The DEC function takes the intermediate tag (V) and ciphertext (C) as the inputs and returns message (M) as the output. If the length of the ciphertext is zero then the message output is a string of zeros. In the case of a non-empty ciphertext the first block of the message is obtained by XORing the first block of the ciphertext with encrypted output of the intermediate tag (V). To generate the i^{th} message block, i^{th} block of the cipher text is eXORed with encrypted value of $(i\text{-}1)^{th}$ ciphertext by fixing it's most significant bit to 1. Figure 4.4 explains the operation of the DEC function.

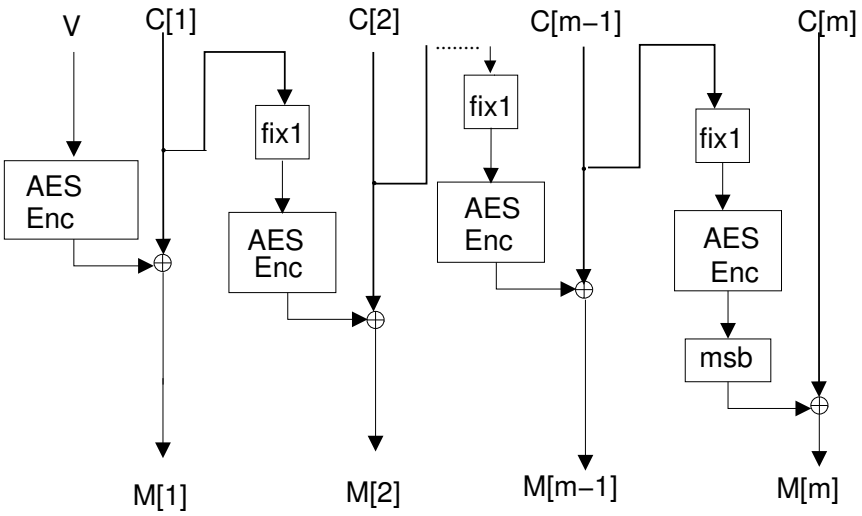


Figure 4.4: DEC Function

4.3 Fullwidth Implementation

Among the three recommended options I have chosen "**aes128n12silcv1"** for my implementation as AES-Enc of AES uses less memory and fewer clock cycles when compared to LED and PRESENT [13] and because using longer nonce will add extra security by reducing the reuse capability.

4.3.1 Datapath Design

1. *Datapath design atthe Input side of the cipher*

   As can be seen from 4.5 the datapath, the input of the AES-Enc consists of a sequence of multiplexers. The data block which is to be passed to the block cipher propagates through the set of multiplexers and arrives at the input of the cipher. Multiplexer **M1** selects between *fed_bdo*, which is given as feedback from the output side and *bdi* which can be either plaintext or associated data. The multiplexer **M2** selects between XORed feedback or fixed output from multiplexer **M1**. The 4×1 multiplexer **Mux_din** selects the input that goes into the AES-ENC core It selects between nonce (IV), tweaked intermediate tag ($g(V)$), intermediate tag (V) and the output of multiplexer **M2**. The inputs are sent into the AES-Enc core based on the subroutines that need to be processed. For instance, if the HASH function needs to

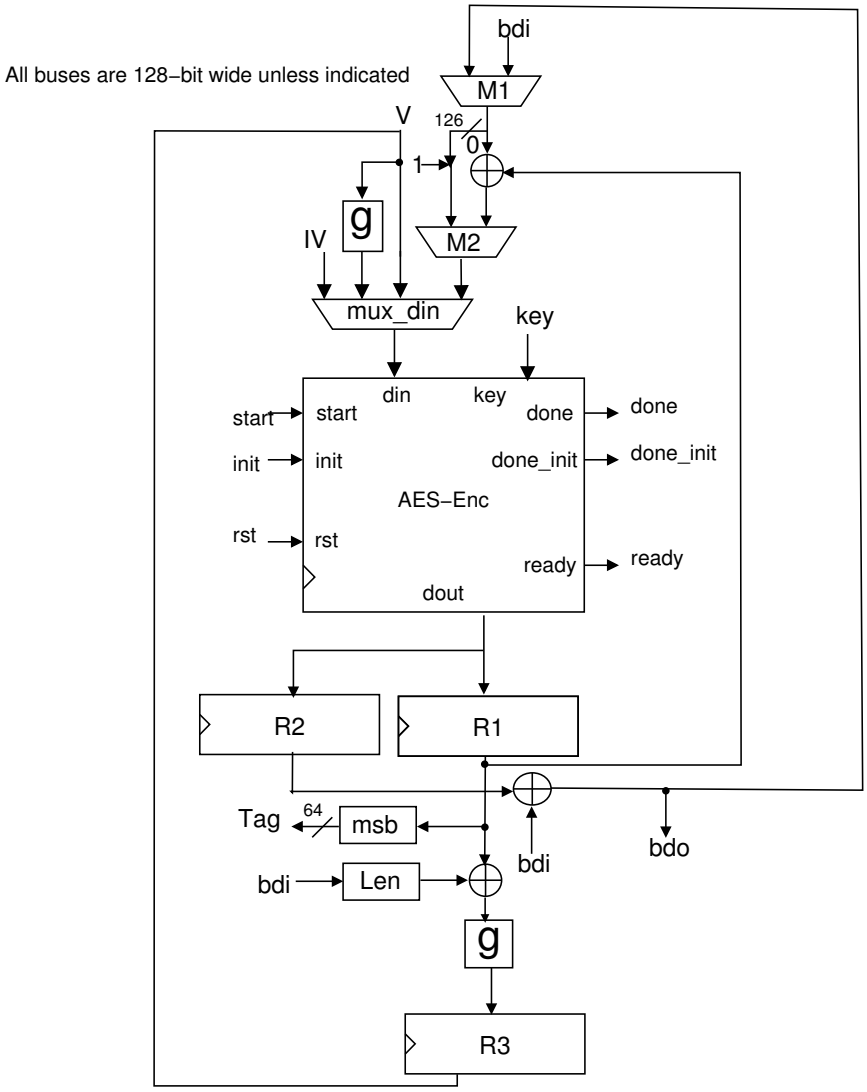


Figure 4.5: Datapath Design of SILC

take place to produce V when the length of associated data is zero then the select signal of the multiplexers are activated in order to send the nonce as an input to AES-Enc core.

2. *Datapath design at the output side of the cipher*

   At the output of the AES-Enc core two registers (**R1**,**R2**) are used to store the output of the AES function. The output of **R1** is then sent as a feedback to the input side. The output of **R1** is also sent to the g function after XORing it with Len (bdi). The output of the g function is then stored into **R3** and sent as a feedback to the input side. The output of **R2** is XORed with *bdi* to generate the cipher text. The ciphertext is then sent as feedback to input side. Tag is generated by taking most significant 64 bits of register *R1* output.

4.3.2 Design of Control Logic

Figure 4.6 shows the toplevel state machine of the SILC fullwidth design. Upon reset the controller enters in the reset state. Then the state is changed to wait state in which the controller waits until the data is given as input. This is indicated by turning the ready signal high. If there is a need of updating the key then the roundkey generation is done. This is done by checking for the "key_needs_update" signal. If this signal is low then the controller goes to the IV_load state. The controller stays in roundkey generation process until the done_rkey signal is high. Once the signal is high the next task is to load IV and process it. In this state the controller waits until the signal iv_ready is high. Once this is done, the controller waits until the bdi_ready signal is high. After the bdi_ready is high the controller checks for bdi_ad signal which indicates that the incoming data is associated data. If the bdi_ad signal is high then the next state is processing associated data. In this state the controller waits for bdi_eot signal which indicates that the associated data has come to an end. If this signal is high then the next process is encryption/decryption. In this state the controller again waits for the bdi_eot. The controller takes 10 clock cycles each to process the associated data and encryption/decryption. After encryption/decryption, the controller waits for bdi_eoi signal which denotes the end of information then the tag generation takes place.

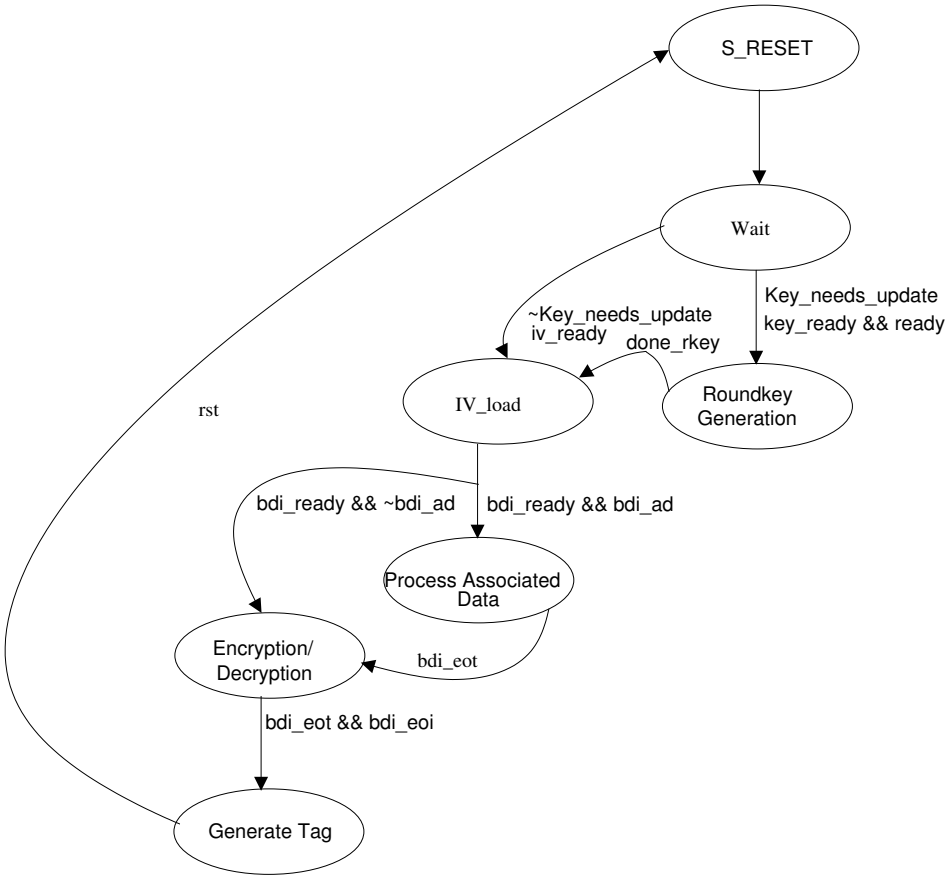


Figure 4.6: Toplevel State Machine of SILC

Figure 4.7 shows the top level diagram of SILC fullwidth design. The names of the signals represent there usage in the datapath. All the enable signals for registers and select signals for the muxes used in the datapath are shown. The "zero_a" and "zero_b" signals that are given as input to the controller are enabled based on the length of the data. The datapath and controller also gets the signals from the pre and post processor of the fullwidth implementation (shown in 3.1. These signals are shown on the top of the block. The outputs from datapath and controller are shown at the bottom of the block.

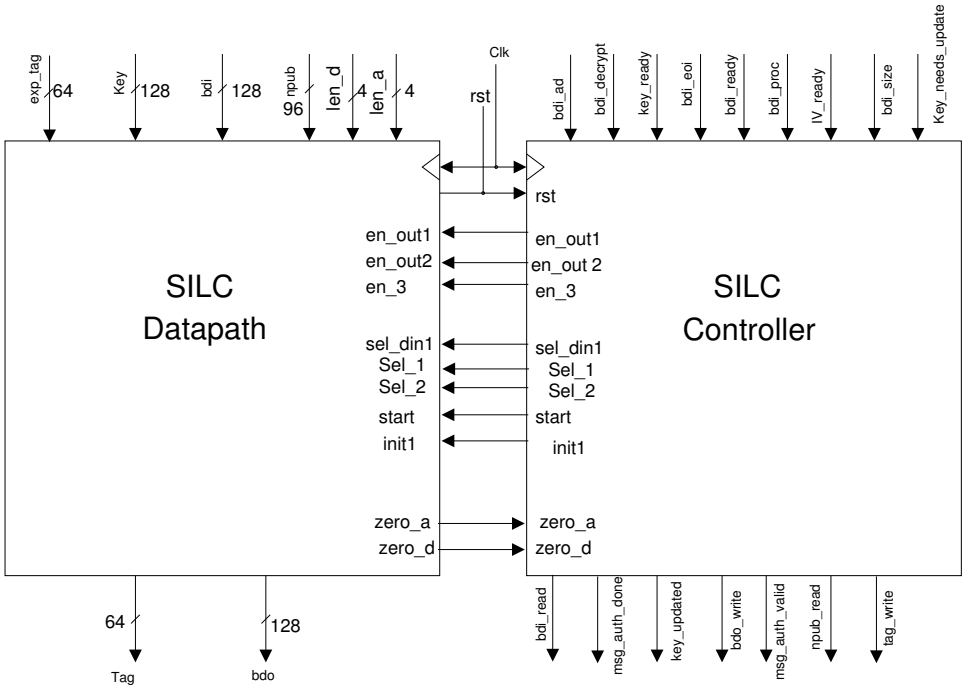


Figure 4.7: Toplevel Structure of SILC

4.4 Lightweight Implementation

The main goal of the optimization is to reduce the area utilization by using a narrow datapath. This section covers the changes made to the design in order to make it a lightweight version.

4.4.1 Datapath Design for Lightweight Implementation

The datapath used for lightweight implementation is 16-bit wide. The 128-bit AES-ENC core is replaced by a compact 8-bit AES-ENC core. We have used the 8-bit version of AES as it uses a very low area (<100 slices). And with the cost of just two clock cycles it can be interfaced with the 16-bit lightweight interface. Major designs changes from the full width version to the lightweight version are given below:

- **At the input side**: As the datapath size is reduced to 8 bits, a register pair (**R0, R1**) and multiplexer pair (**M1, M5**) is used to select between first and second byte of public and secret

data. When using the compact AES core, key is needed to be given to the *datain* bus directly so, an additional 8bit 4×1 is used to select between key, data and other strings that indicates the type of input. Width of all the multiplexers was reduced to 8bits.

- **At the output side**: The 128-bit registers **R1** and **R2** are replaced with 64×6 DRAMs. The register **R3** which stores the intermediate tag(V) is removed and this operation is now handled by using an additional multiplexer (**M6**) at the top of second DRAM. A counter is added to the design in order to generate addresses for the DRAMs. The g function operation is performed by using a 8bit register (**R2**) at the end of first DRAM. An additional multiplexer (**M7**) is used to selected between tag and ciphertext. As the output of the interface is 16 bits wide we need an additional register (**R3**) at the output side to store the first byte coming out of the multiplexer.

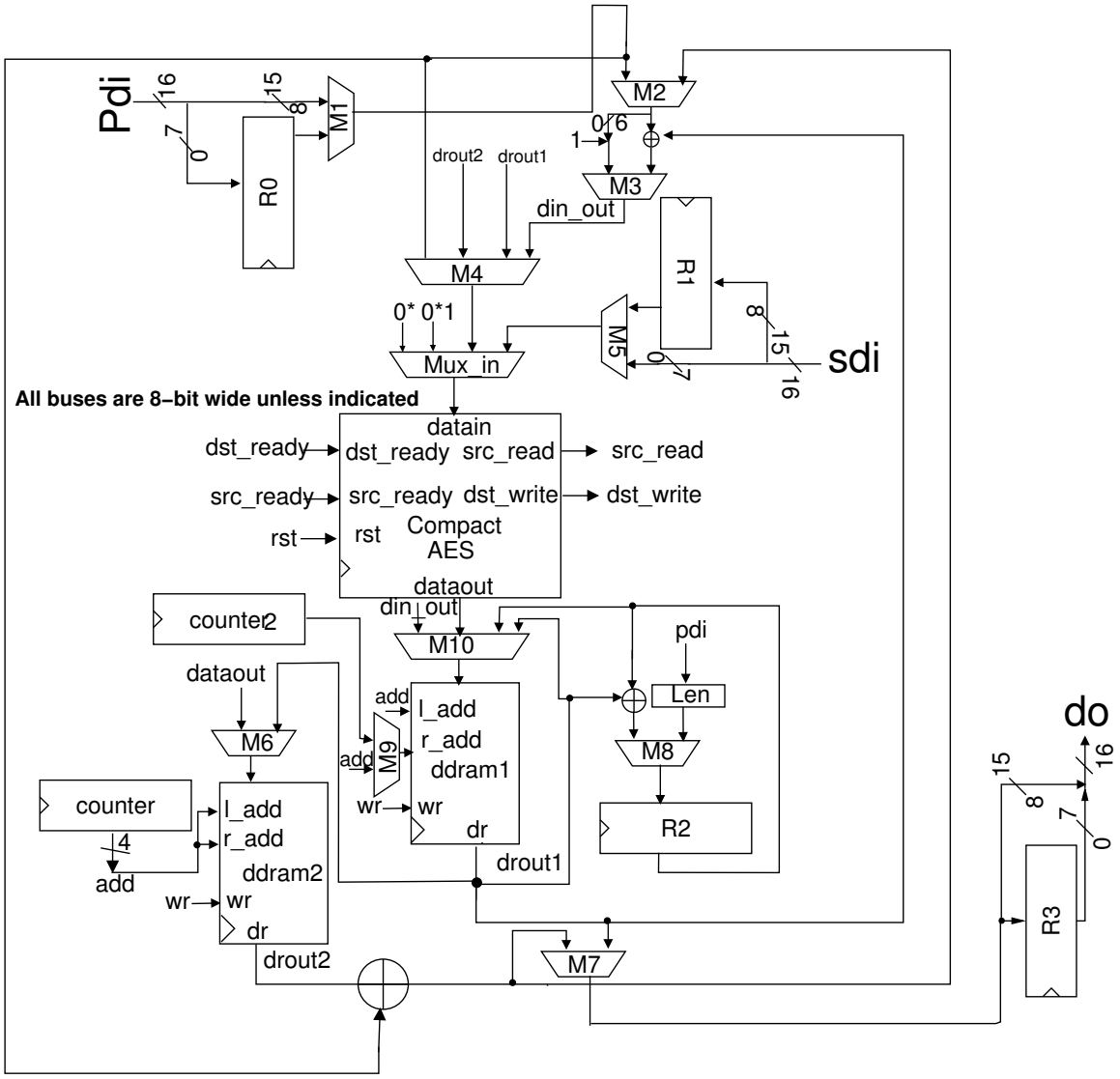


Figure 4.8: Lightweight Design of SILC

4.4.2 Design of Controller

The toplevel flow of controller of the lightweight design is shown in figure 4.9. The flow is similar to that of the fullwidth design but the signals that controller check are different.The controller stays in the "wait" state until the key is ready which means it waits until the sdi_valid signal is high In the "wait" if the get_key is high then the roundkey generation takes place if not the IV is processed directly. As the public data comes on a single bus in the lightweight interface, the signal pdi_type is

used to distinguish between the type of data coming in. The end of type signal "eot" is checked while processing each type of data. Once the encryption/decryption is completed the controller checks for last_word_in signal and if this signal is high the next process is to generate the tag.

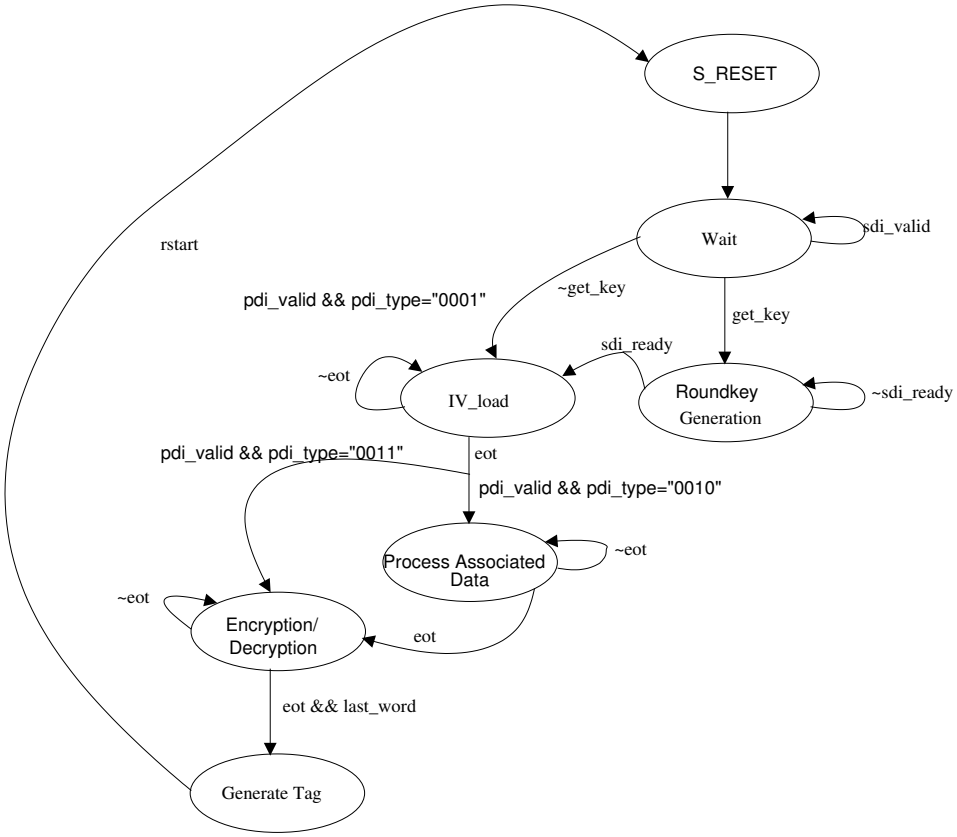


Figure 4.9: Toplevel State Machine of Our Lightweight Design of SILC

The toplevel structure of Our SILC lightweight design is shown in figure 4.10. The signals exchanged between datapath and controller are shown. The controller for Our lightweight design requires more clock cycles compared to the full width controller. We use counters to count the number of bytes that are being encrypted by the AES-core.

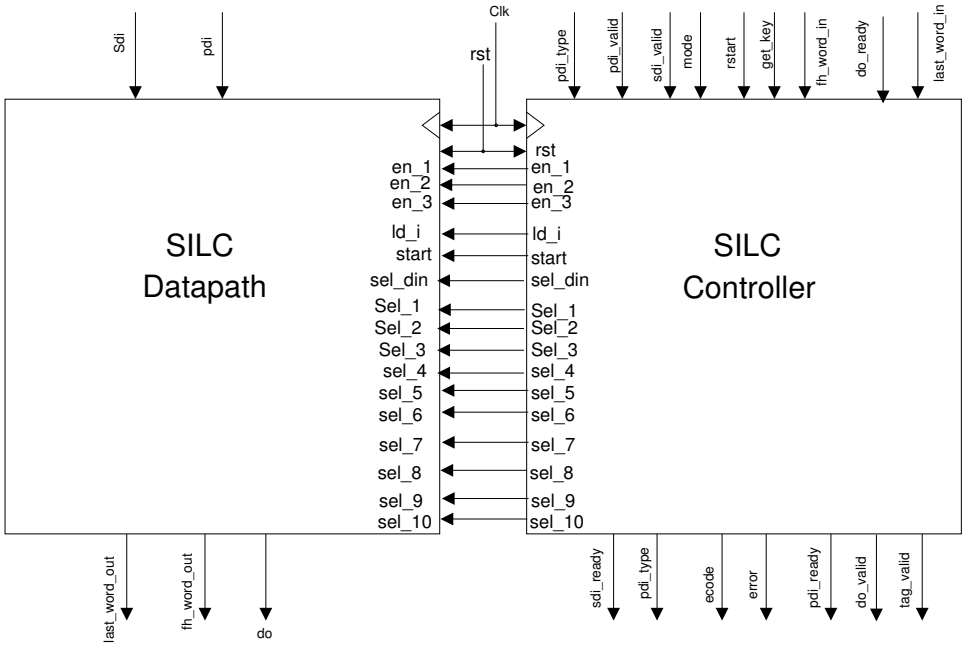


Figure 4.10: Toplevel Structure of Our SILC Lightweight Design

Chapter 5: Joltik

This chapter focuses on the implementation of Joltik. We first discuss the tweakable block cipher, Joltik-BC and later explain Joltik and it's implementation.

5.1 Introduction

Joltik is a lightweight authenticated encryption algorithm based on the tweakable block cipher Joltik-BC which is AES like. Joltik has two main variants which differ in the way they allow the usage of the nonce (N):

- **Joltik nonce-respecting** ($E^{\neq}$ and $D^{\neq}$): In this variant the user is not allowed reuse the same nonce for encrypting with the same key.
- **Joltik nonce-misuse resistant** ($E^{=}$ and $D^{=}$): In this variant the user can reuse the nonce with the same key.

5.1.1 Features

Joltik provides good security with only a single call to the block cipher per message [31]. It is simple to design and analyze both the internal tweakable block cipher and the authentication mode. Joltik can resist side-channel attacks with the same techniques used for AES.

5.2 Joltik-BC

Joltik-BC is a tweakable 64-bit block cipher with 128-bit key. It takes an additional input named as tweak (T) along with message (Plaintext(P) or Ciphertext (C)). Symbolically, the encryption is denoted as E_K(T,P)=C and the decryption is denoted as E_K^{-1}(T,C)=P. The design of Joltik-BC is an iterative substitution-permutation network similar to that of AES The state is a 4×4 matrix of nibbles (4-bit word). The number of rounds r is 24 for Joltik-BC-128 (size of key+tweak=128

bits) and 32 for Joltik-BC-192 (size of key+tweak=192 bits). Any round in Joltik-BC has four transformations that are applied to the internal state:

- **AddRoundTweakey**- A 64-bit round subtweakey (defined in 5.2.3) is XORed to the internal state.
- **SubNibbles** - Apply 4-bit S-Box(defined below in 5.2.1) to the internal state.
- **ShiftRows** - Rotates the 4-nibble *i-th* row left by i positions.
- **MixNibbles** - Multiplies the internal state with the constant MDS matrix (defined below in 5.2.2)

After the above transformations a final **AddRoundTweakey** is applied to get the ciphertext.

5.2.1 S-box

The 4-bit S-box used in Joltik-BC is the one selected for the Piccolo block cipher [46]. It is defined by the following table 5.1.

Table 5.1: S-Box Used in Joltik-BC

nibble in	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
nibble out	14	4	11	2	3	8	0	9	1	10	7	15	6	12	5	13

5.2.2 MDS Matrix

The MDS (M and M^{-1}) matrix used in Joltik-BC is involutary. The matrix is shown below:

$$\begin{bmatrix} 1 & 4 & 9 & 13 \\ 4 & 1 & 13 & 9 \\ 9 & 13 & 1 & 4 \\ 13 & 9 & 4 & 1 \end{bmatrix}$$

The decryption operation in Joltik-BC is similar to that of encryption but inverse transformations are applied in a reverse order.

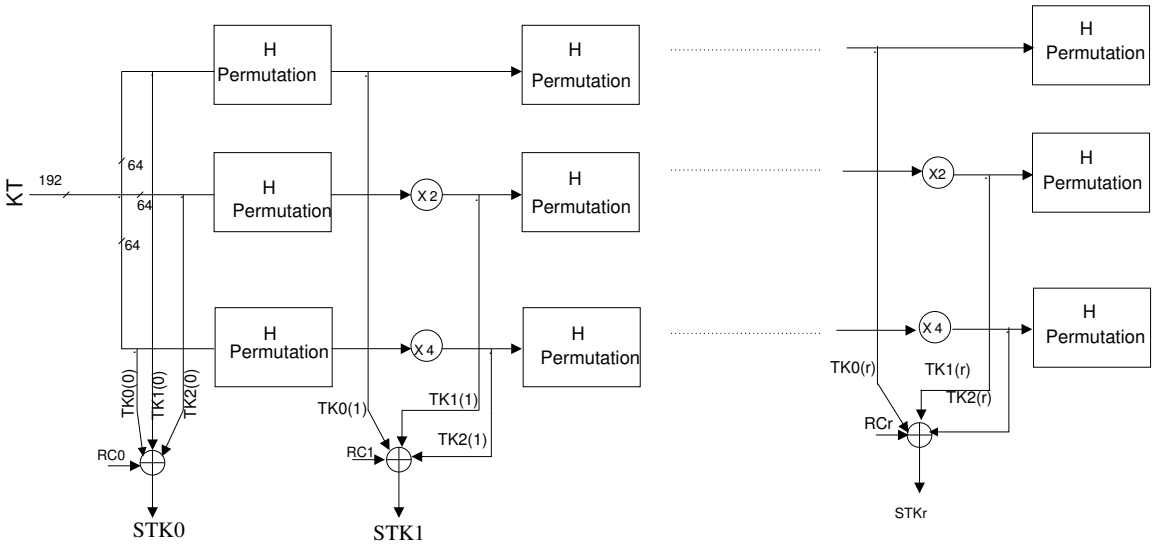


Figure 5.1: Key Scheduling Algorithm of Joltik-BC 192

5.2.3 Generation of Subtweakeys

The subtweakey at round i (STK_i) is defined below:

for Joltik-BC-128:

$\text{STK}_i = \text{TK}_i^1 \oplus \text{TK}_i^2 \oplus \text{RC}_i$

for Joltik-BC-192:

$\text{STK}_i = \text{TK}_i^1 \oplus \text{TK}_i^2 \oplus \text{TK}_i^3 \oplus \text{RC}_i$.

Where TK_i^1, TK_i^2, TK_i^3 are tweakeys produced by the key scheduling algorithm (KS(W,α)) shown in figure 5.1 The algorithm takes a 64-bit word W and a constant α as inputs and gives subkeys as the output sequentially by applying a nibble permutation(h)(defined below), and a finite field multiplication g. Subkey of i^{th} round is defined as $\text{TK}_i = \text{g}(\text{h}(\text{TK}_{i-1}))$. The h permutation used in the key scheduling algorithm is shown in table 5.2

Table 5.2: H Permutation in Joltik-BC

in	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
out	1	6	11	12	5	10	15	0	9	14	3	4	13	2	7	8

Key and tweak are concatenated to form KT (192 bits in the case of Joltik-BC-192). The outputs of the key scheduling algorithm can be defined as follows:

TK_i^1 = KS(W_1,1), where W_1 are the first most significant 64 bits of KT.

TK_i^2 = KS(W_2,2), where W_2 are the second most significant 64 bits of KT.

TK_i^3 = KS(W_3,4), where W_3 are the third most significant 64 bits of KT.

The round constants used in the key scheduling algorithm are similar to the constants used in LED cipher [20].

5.3 Encryption and Decryption

The encryption of Joltik starts with processing of associated data to produce "Auth" and then message processing is done to produce the ciphertext and tag.

Processing of Associated Data

Associated data is encrypted by concatenating tweak (T) as Nonce, "0010" and block count to form the tweak (T). Outputs of all the encryptions are XORed together to get the intermediate "Auth", which is later used for generation of tag. Figure 5.2 explains the algorithm.

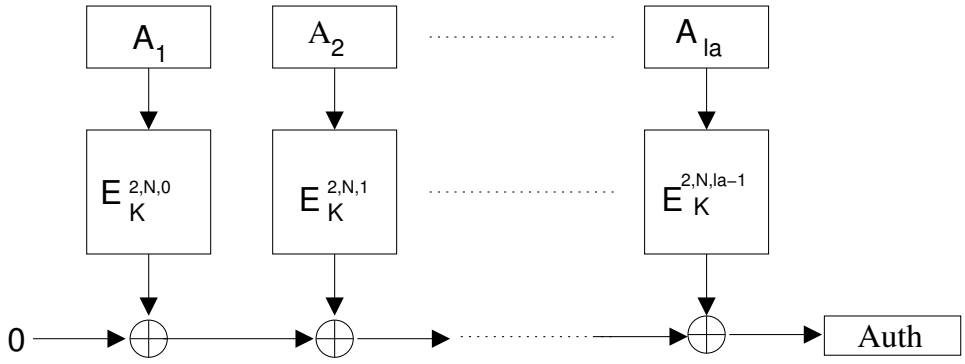


Figure 5.2: Associated Data Processing Without Padding

In the case where the length of the associated data is not a multiple of the block size, padding takes place. Joltik uses $pad10^*$ function that applies 10^* padding such that total length of the data

is equal to multiple of block size (n). While encrypting the padded block the tweak is formed by concatenating Nonce, "0110", block count

$$pad(10^*)(\mathrm{X})=\mathrm{X}\|1\ \|0^{n-|X|-1}$$

The figure 5.3 shows the algorithm for processing associated data with padded last block.

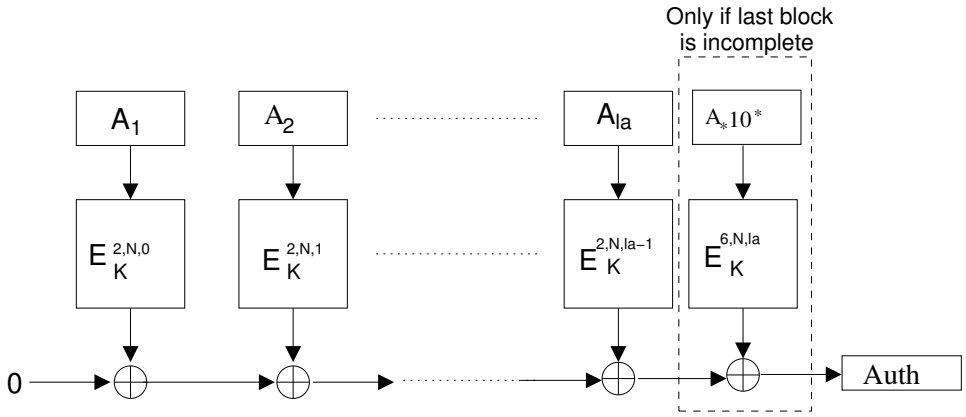


Figure 5.3: Associated Data Processing with Padding

5.3.1 Message Processing

A message is encrypted by taking tweak as concatenation of"0000 || Nonce|| block counter" to generate the ciphertext. Tag is produced by XORing the "Auth", which is generated by processing associated data, with the encrypted output of the checksum. Checksum is computed by first initializing to zero and by XORing the message bits. Figure 5.4 shows the algorithm.

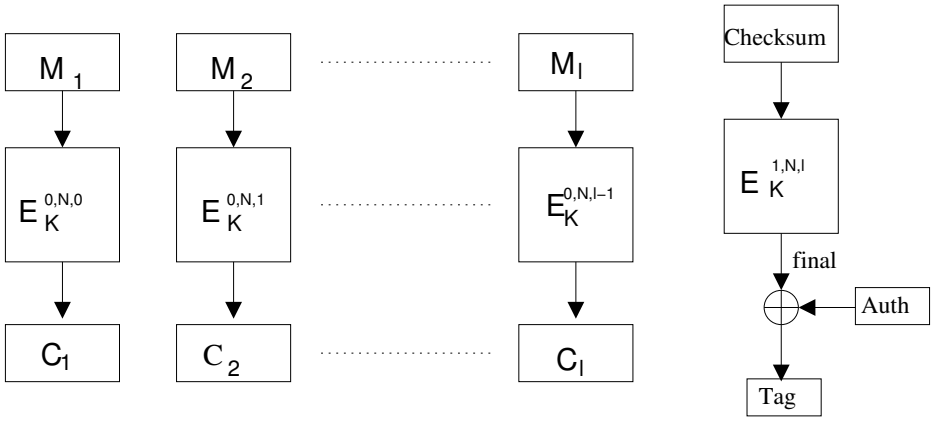


Figure 5.4: Message Processing Without Padding in Joltik

Similar to associated data processing in the case where the length of the message is not a multiple of the block size padding is done. In order to produce the last ciphertext block, the padded message is XORed with "final", which is produced by encrypting an empty string of length equal to block size by taking "0100 ‖ Nonce‖ block counter" as the tweak. Tag is obtained by XORing the *"Auth"* with *"final"* which is produced by encrypting checksum by taking "0101 ‖ Nonce‖ block counter" as the tweak input. Figure 5.5 shows the algorithm.

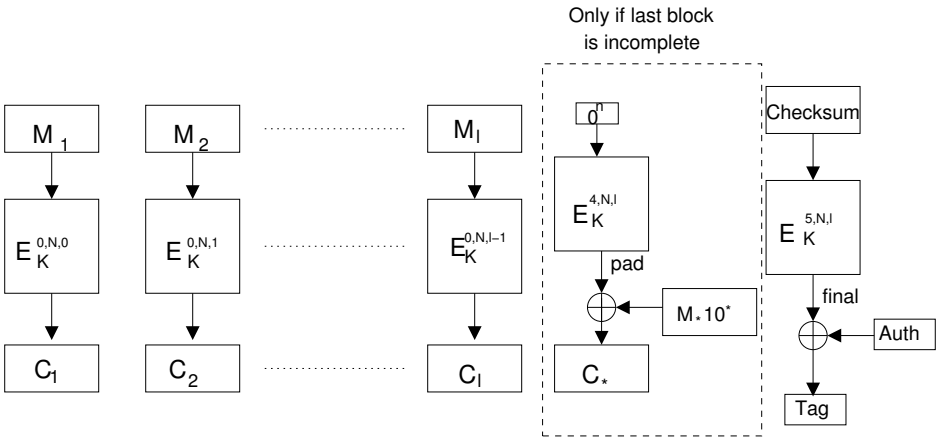


Figure 5.5: Message Processing With Padding in Joltik

5.4 Fullwidth Implementation

Joltik is designed around Joltik-BC core. It is the most basic implementation which is neither optimized for area nor throughput. As Joltik is inverse free a single Joltik-BC is used for both encryption and decryption and a decrypt signal selects between the two operations.

5.4.1 Datapath Design

Datapath design is explained below in two parts

Datapath Design at the Input Side of the Cipher

As shown in figure 5.6, the datapath at the input of the block cipher consists of a sequence of multiplexers and registers. The data block which is to be passed to the block cipher propagates through the multiplexer **mux_din** and arrives at the input of the cipher. The tweak which is given as the input to the Joltik-BC is not same at all times, it changes based on the operation being performed. So we require a series of two 2×1 (**M1**, **M2**) and one 4×1 (**M3**) 4-bit multiplexers which fetches 4-bit strings and concatenate with nonce and counter value to update the tweak.The *checksum* is generated by XORing message with an empty string and stored in the register **Reg_C** and updated in parallel. The *checksum* is fetched as input into the Joltik-BC core when the last block of the AD/message has arrived. The input is processed differently in the case, where the message length is not a multiple of the block size.

Datapath Design at the Output Side of the Cipher

At the output side the register **Reg_A** is used to store *Auth*. It is first initialized to a string of zeros and then updated by XORing the output of encryption of associated data with the string of zeros. *Final*, which is obtained by encrypting *checksum* is used to generate *tag* by XORing it with *Auth*. *Pad* is produced by encrypting an empty string of zeros of length equal to block size. In the case where the message length is not a multiple of the block size, the last ciphertext block (C_*) is generated by XORing *pad* with input (bdi_*). *pad* is produced by encrypting a string of zeros of length equal to block size.

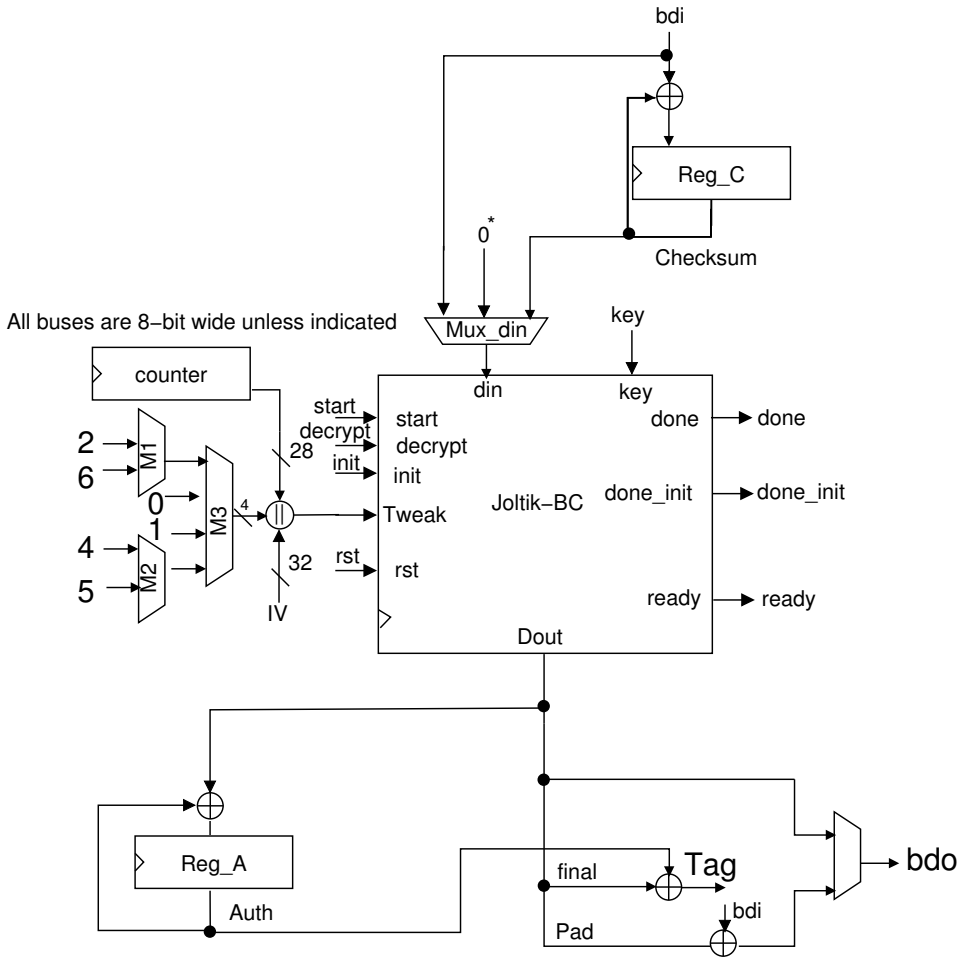


Figure 5.6: Datapath Design of Joltik

5.4.2 Design of Control Logic

Figure 5.7 shows the top level state machine of our fullwidth Joltik design. Upon reset the controller enters into the Reset state where in all the registers are reset. Then the state is changed to wait state and in which the controller waits until the data is given as input and this is shown when the ready signal is high. If there is a need of updating the key then the roundkey generation. This is done by checking for the "key_needs_update" signal. If this signal is low then the controller goes to the "IV_load" state. The controller stays in subtweakey generation process until the "done_rkey" signal is high. Every time a new data comes in the tweak must be updated so the "bdi_ready" signal is checked before generating the subtweakey. Once the signal is high the next task to load

IV and process it. In this state the controller waits until the signal "iv_ready" is high. After this, the controller checks for "bdi_ad" signal which indicates that the incoming data is associated data. If the "bdi_ad" signal is high then the next state is "process associated data". In this state the controller waits for "bdi_eot" signal which indicates that the associated data has come to an end. If this signal is high then the next process is encryption/decryption. In this state the controller again waits for the "bdi_eot". The controller takes 32 clock cycles each to process the associated data and encryption/decryption. After encryption/decryption, the controller waits for "bdi_eoi" signal which denotes the end of information then the tag generation takes place.

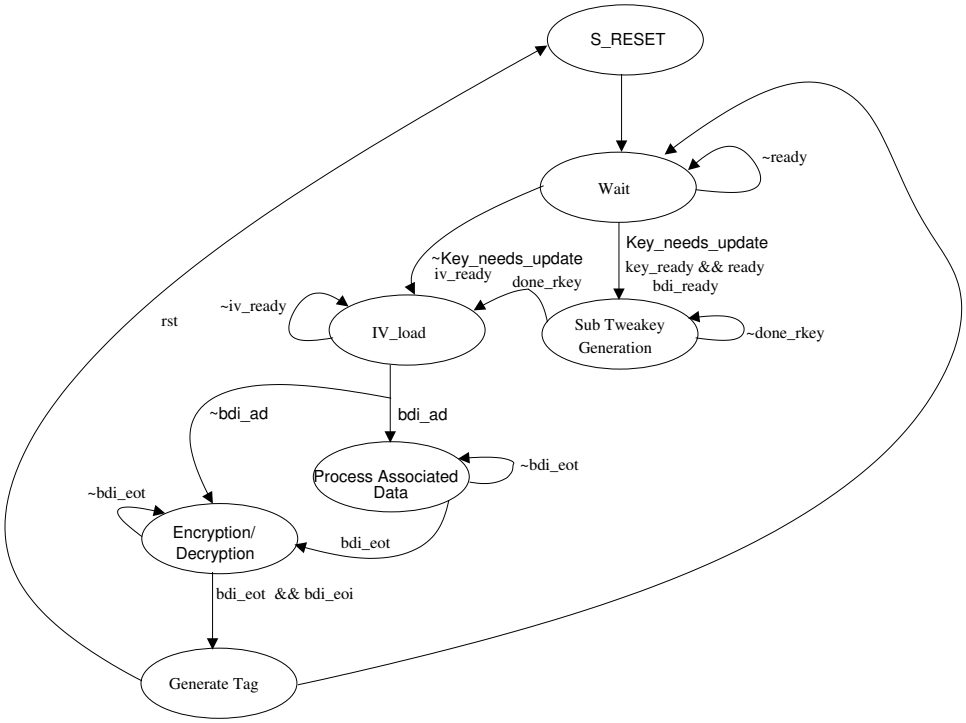


Figure 5.7: Toplevel State Machine of Joltik

The signals exchanged between the datapath and controller are shown in the figure 5.8. The names of the signals represent the usage of the signal in the datapath. The controller provides the select signals "Sel_1", "Sel_2", "Sel_3" for the multiplexers used in the datapath to update the tweak according to the operation being performed. Other select signals ("sel_din", "sel_dout") are given for the multiplexers at the input inside and output side of the cipher. The enable signals "en_a"

and “en_c” are given to the registers that store the checksum and auth values. The controller checks for the delimiter signals “bdi_eoi” and “bdi_eot” which denote the end of data and end of type. Upon checking for the delimiter signals the controller provides the select signals and enable signals as required to the datapath core.

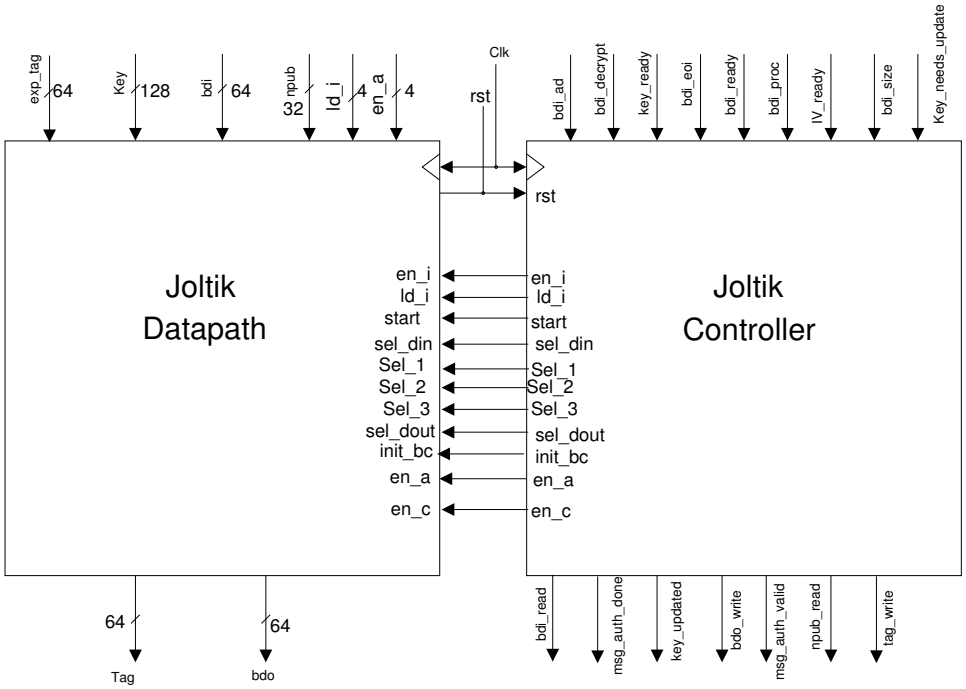


Figure 5.8: Toplevel Diagram of Joltik

5.5 Lightweight Implementation

This section covers the lightweight design of Joltik. First, we will discuss the lightweight version of the tweakable block cipher Joltik-BC. Later, we will discuss the design of Joltik authenticated encryption cipher which is built on the Joltik-BC.

5.5.1 Lightweight Joltik-BC

The lightweight design can be explained by splitting it into four transformations that are applied to the internal state:

Shiftrows

Shiftrows is done by adjusting the address of the DRAM by using two 4×1, a 2×1 multiplexer and a register. The output of the register is given as an address to the DRAM that stores the data input.

SubNibbles

The 4-bit S-box is applied to the output of the first DRAM.

MixNibbles

The MixNibbles operation is done by multiplying the rows with 1, 4, 9, 13 and XORing them together. For this operation we use four 4×1 multiplexers and 4 registers (**R1,R2,R3,R4**).

Key Scheduling

Key Scheduling is done by using 2 levels of DRAMs. The first level of DRAMs store the original key where as the second level stores the key after the g multiplication. The address of the second level DRAMs are H permuted.

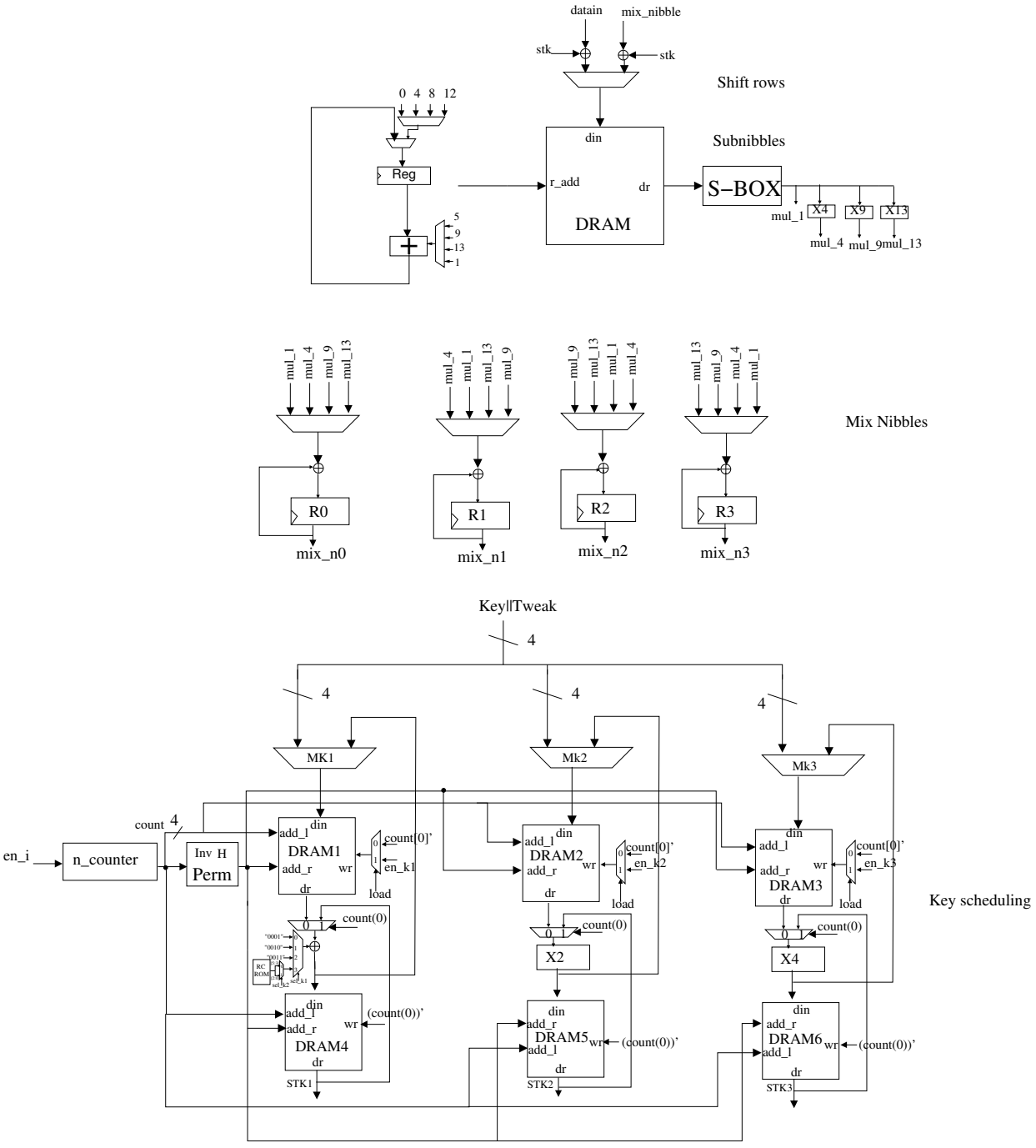


Figure 5.9: Optimized Datapath Design of Joltik-BC

5.5.2 Optimized Datapath for Lightweight Implementation

The datapath is designed to optimize the area. Figure 5.10 shows the design. The changes made to the fullwidth datapath are mentioned below:

At the Input Side

- The register **Regc** which stores the checksum is replaced with a DRAM.
- The width of **mux_din** is changed from 64-bit to 4-bit.
- As the datapath size of Joltik-BC is 4-bit and the inputs are 16 bits wide, three 4-bit registers **R1,R2,R3** are used to store the nibbles and a 4-bit 4×1 multiplexer (**M4**) is used to select between the nibbles.
- An additional 4-bit 4×1 multiplexer (**M5**) is used to select between key, IV, state string, and counter.

At the Output Side

- Three 4-bit registers **R4,R5,R6** are used to store the encrypted nibble and concatenate the ouputs to get a 16-bit output.
- The register **Rega** is replaced with a 16×4 DRAM (**DRAM2**).
- As the interface has a single ouput port an additional multiplexer (**M6**) is added to select between tag and ciphertext.

5.5.3 Controller Design

The toplevel flow of the controller for lightweight Joltik design is shown in figure 5.11. The flow is similar to that of the fullwidth design but the signals that controller check are different. Upon reset the controller enters into the Reset state where in all the registers are reset. The controller stays in the "wait" state until the key is ready which means it waits until the "sdi_valid" and "pdi_valid" signal is high because we need to have IV before generating subtweakeys. In the "wait" state if the "get_key" is high then the subtweakey generation takes place if not the IV is processed directly. As in the lightweight interface, the public data comes on a single bus, the signal "pdi_type" is used

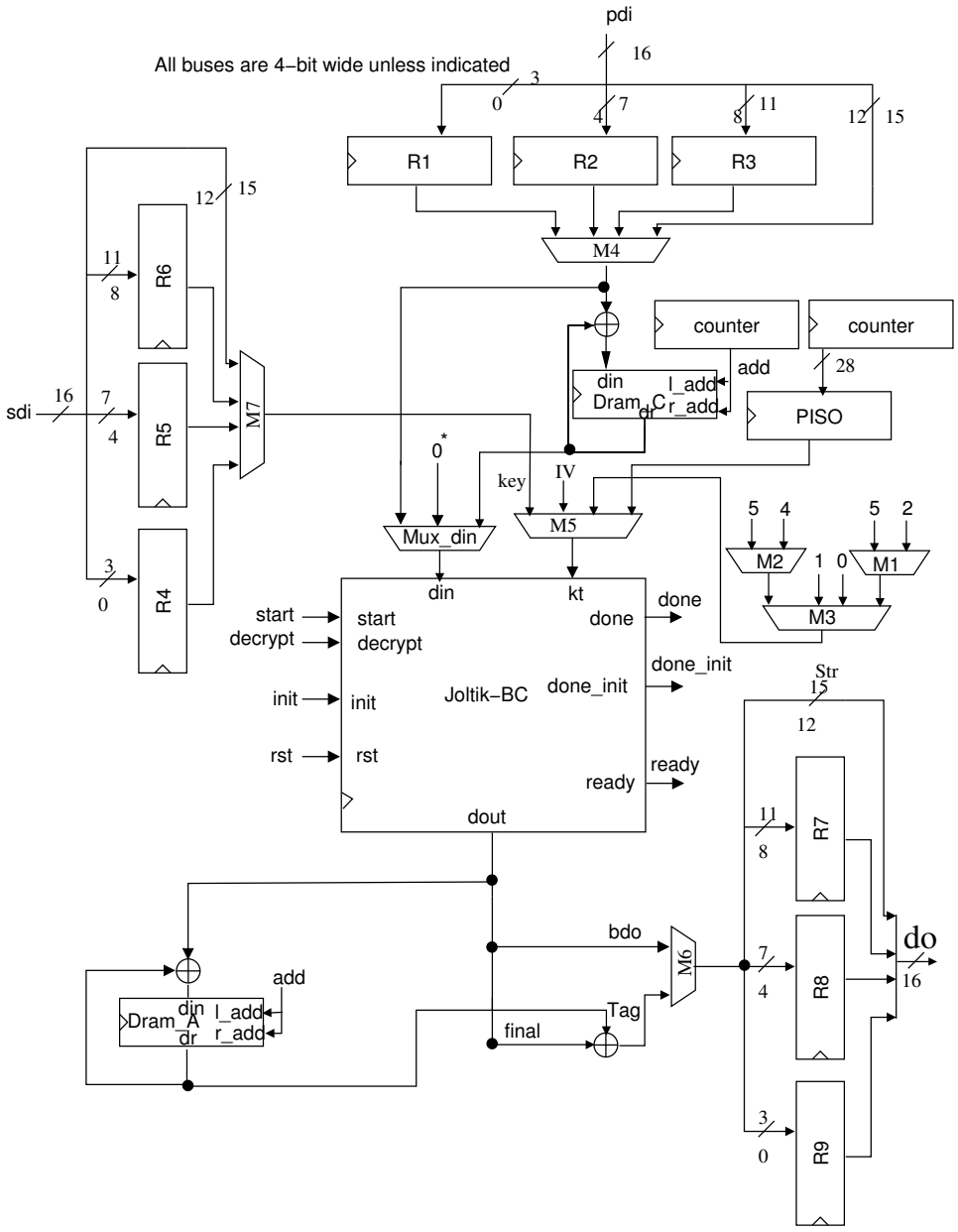


Figure 5.10: Lightweight Datapath Design of Joltik

to distinguish between the type of data coming in. The end of type signal "eot" is checked while processing each type of data. Once the encryption/decryption is completed the controller checks for the "last_word_in" signal and if this signal is high the next process is to generate the tag.

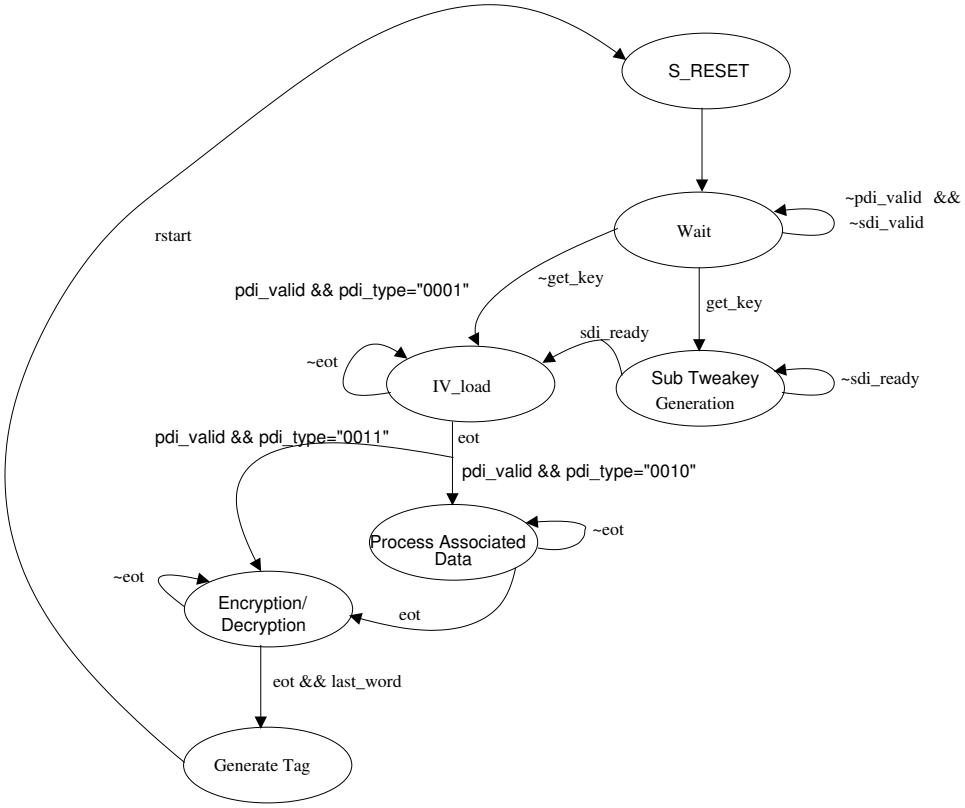


Figure 5.11: Toplevel State Machine of Our Lightweight Joltik Design

The top level architecture of the controller is shown in figure 5.12. When compared to the fullwidth design, the lightweight design has larger number of registers and multiplexers. The signals that are exchanged between the controller and the datapath are shown. All the enable signals and select signals are given to the datapath based on the operation required to be performed.

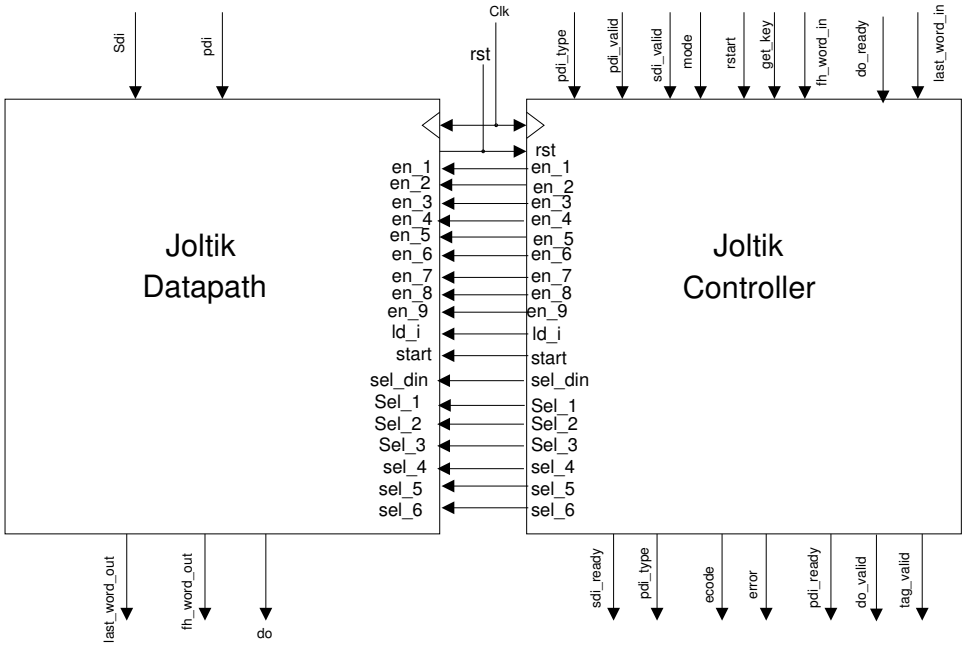


Figure 5.12: Toplevel Structure of Joltik

Chapter 6: ACORN: A Lightweight Authenticated Cipher

The last candidate we implemented is ACORN. In this chapter we first explain the functions that build the structure for ACORN. Later, we discuss the hardware implementation of ACORN.

6.1 Introduction

6.1.1 Features

ACORN is a bit-wise authenticated cipher [50] which means, one bit of message is processed in one step there by benefiting the lightweight hardware. It allows parallel computation, so high speed hardware and software implementation is possible. The hardware cost of ACORN is low and is very efficient. The length of message and data is not needed in ACORN which means it has a fixed padding and doesn't need padding to a multiple of block size. This reduces the cost of hardware.

6.1.2 Functions Used in ACORN

Boolean functions

There are two boolean functions $maj(x, y, z)$ and $ch(x, y, z)$ explained in figure 6.1

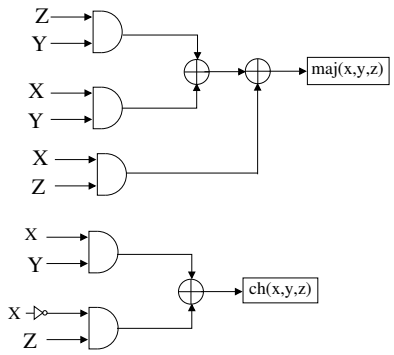


Figure 6.1: Boolean Functions of ACORN

Keystream Generation Function (KSG)

This function is used to generate a keystream bit at every step of ACORN. It takes the 293-bit state (S) as input and gives a single keystream bit as output. Figure 6.2 shows the operation of KSG function.

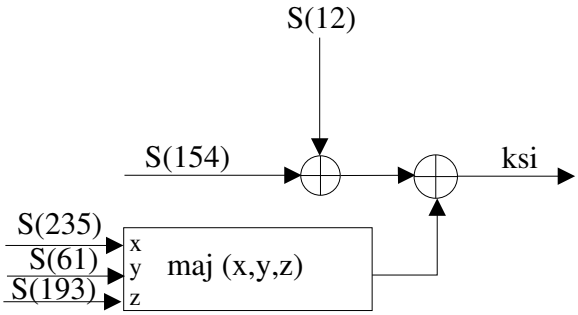


Figure 6.2: KSG function

Feedback Bit Generation (FBK) Function

The FBK function takes the present state (S_i) and control bits (C_a, C_b) as inputs and returns feedback bit f_i as the output at each step of the operation. Figure 6.3 shows the operation of FBK.

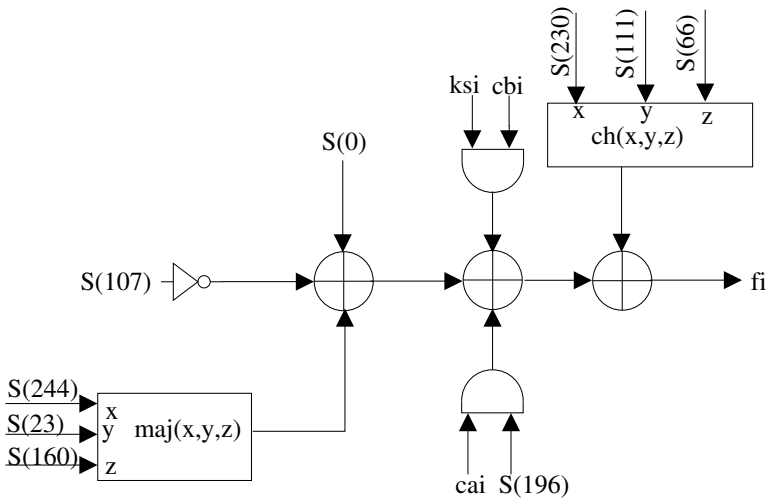


Figure 6.3: FBK Function

State Update128 Function

The state update function takes present state (S_i), data bit (m_i) and control bits (C_a, C_b) as inputs and generates the next state (S_{i+1}) as the output. Figure 6.4 shows the block diagram of stateupdate128 function.

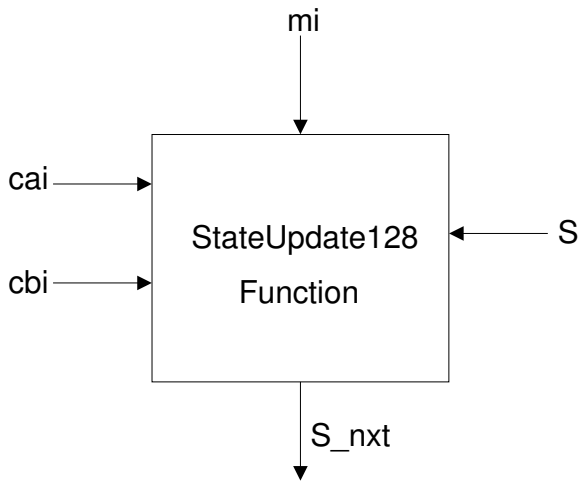


Figure 6.4: Block Diagram of State Update Function

The feedback bit (f_i) is generated by using the FBK function. This feedback bit is then XORed with databit (m_i) to generate the 293^{th} bit of the next state. Figure 6.5 shows the operation of state update function.

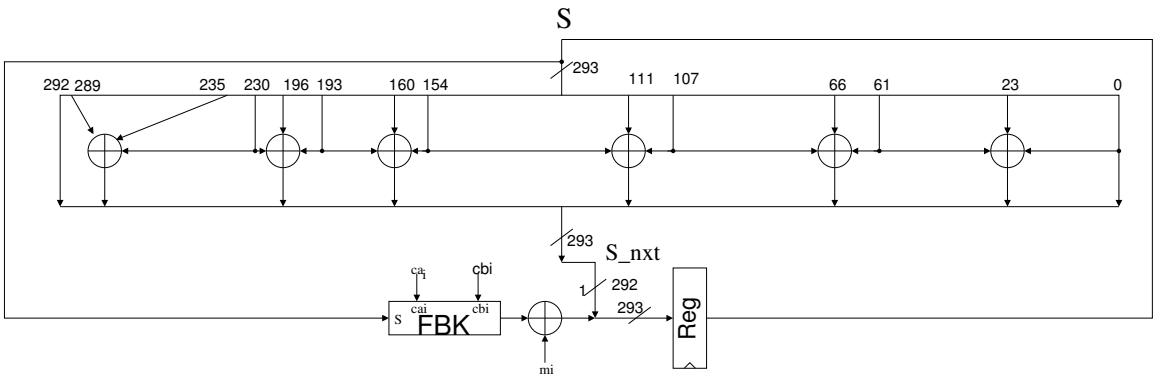


Figure 6.5: State Update Function

6.2 Encryption and Decryption

6.2.1 The Initialization

The initilization step includes loading of Initialization Vector (IV) and key (K) into the state and updating the state 1792 times. The initial state is set to an empty string of 293 bits. The inputs to the state update function during initialization are tabulated in table 6.1.

Table 6.1: Message and Control Bits in Initialization Process

i	-1792 to -1665	-1664 to -1537	-1536	-1535 to 0
m	K_i	IV_i	$K_0 \oplus 1$	$K_{i \, mod \, 128}$
C_a	1	1	1	1
C_b	1	1	1	1

6.2.2 Processing the Associated Data

This step follows after the initialization step and uses the associated data to update the state. Even when there is no associated data the state is updated 256 times. The inputs to the stateupdate function for processing associated data are show in table 6.2.

Table 6.2: Message and Control Bits for Processing Associated Data

i	0 to adlen-1	adlen	adlen+1 to adlen+127	adlen+128 to 255
m	AD_i	1	0	0
C_a	1	1	1	0
C_b	1	1	1	1

6.2.3 The Encryption

At each step of the encryption a plaintext bit is used for updating the state and producing the ciphertext bit. The inputs to the state update function in this step are shown in 6.3.

Table 6.3: Message and Control Bits for Encryption

i	adlen+256 to adlen+256+msglen-1	adlen+256+msglen	adlen+256+msglen+1 to adlen+383+msglen	adlen+384+msglen to adlen+511+msglen
m	message_i	1	0	0
C_a	1	1	1	0
C_b	1	1	1	1

After updating the state, a keystream is generated over the state using KSG128 function and ciphetext bit (c_i) is produced by XORing plaintext bit(p_i) with a keystream bit

$$c_i = p_i \oplus \text{KSG128}(S_i)$$

6.2.4 The Finalization

This step is used for generating the authentication tag bit T. In this step the the state is updated 512 times. The inputs to the state update function at this stage are tabulated in 6.4

Table 6.4: Message and Control Bits for the Finalization

i	adlen+msglen+212 to adlen+pclen+1279
m	0
C_a	1
C_b	1

After the state update, keystream bits are produced by running the KSG128 function over the state. The tag is generated by concatenating the last t bits of the keystream, where t is the length of the tag.

$$\text{ks}_i = \text{KSG128}(S_i)$$

$$\text{Tag(T)} = ks_{adlen+msglen+1535-t+1} \| ks_{adlen+msglen+1535-t+2} \| \ldots \| ks_{adlen+msglen+1535-t+t}$$

6.2.5 Decryption and Verification

Decryption and verification are similar to that of encryption and tag generation. If the verification of the tag fails, the ciphertext and the new tag are not given as output.

6.3 Fullwidth Implementation

The implementation of ACORN is simple when compared to the other two candidates as it is based on a strem cipher. The design is wrapped around the stateupdate128 (6.1.2) function.

6.3.1 Datapath Design

The design is 8-bit wide, but the inputs from the interface are 128 bit wide so we use three parallel in serial out shift registers to load the 128 bit inputs in parallel and shift 8 bits at a time serially. The design also has two cascaded 8-bit multiplexers, a 2×1 multiplexer to select between key and iv and a 4 ×1 multiplexer which selects between block data input (Message/Associated Data), key/IV, "0" and "1". The output of the multiplexer is then split into eight single bits and then given as the input message bit to eight stateupdate128 functions (explained earlier in 6.1.2). Using 8 state update functions will pipeline the design. The stateupdate function also takes control bits C_a and C_b as inputs. The value of these control bits can be either 0 or 1 depending on the operation. Additionally a 293-bit register (**Reg_state**) is used to store the next state. The output of state update functions is given as input to the next which is then given as the input to produce the further next state. The present state of every state update function is also given as input to 8 KSG128 functions (explained earlier in 6.1.2) to generate the key stream bit, which then combined together to get a keystream byte. The key stream byte is sent serially into a Serial-In-Parallel-Out(SIPO) shift register and given as 128-bit tag output. The output of SIPO is then XORed with the plaintext to give the 128-bit ciphertext. Figure 6.6 shows the datapath design of ACORN.

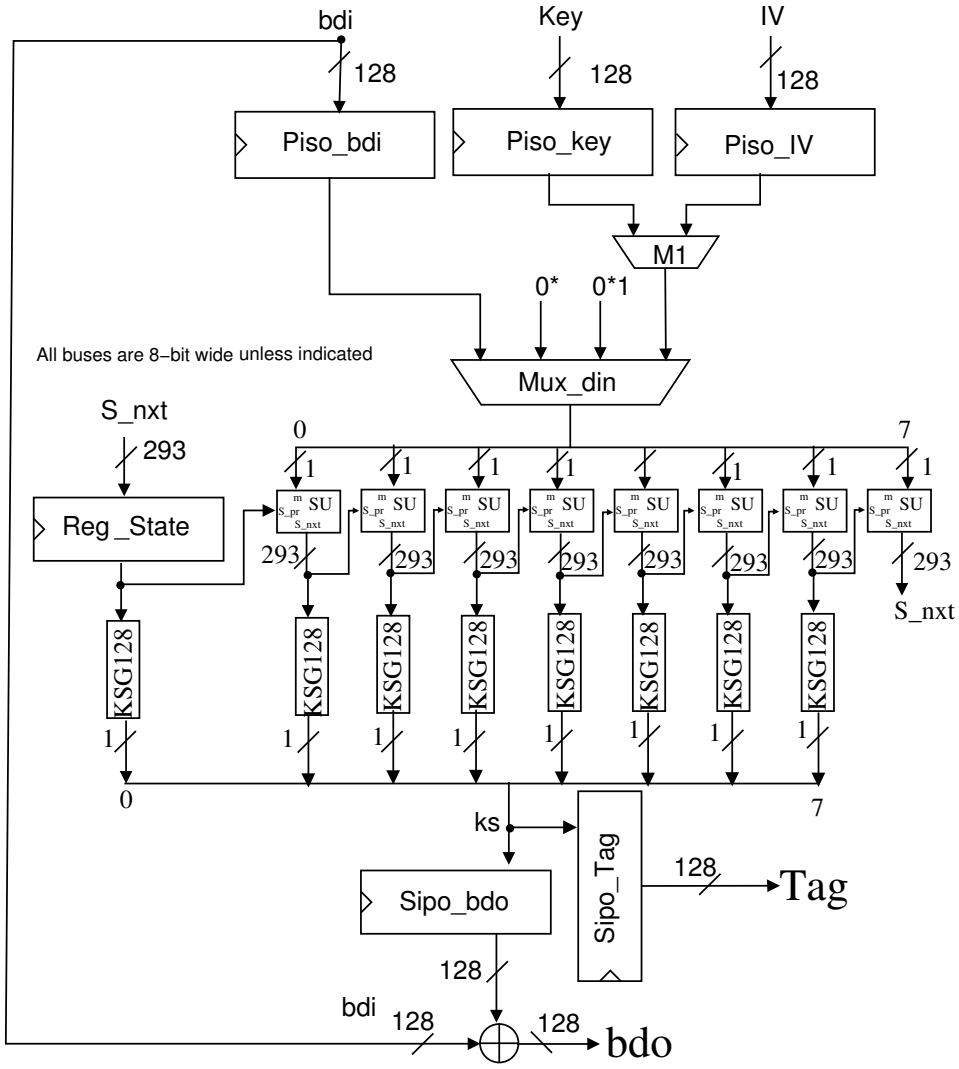


Figure 6.6: Datapath Design of ACORN

6.3.2 Design of Control Logic

The controller of ACORN has separate states for loading and processing the data. Figure 6.7 shows the top level state machine of fullwidth Acorn design Upon reset the controller stays in the reset state. Then the state is changed to wait state and in this state the controller waits until the data is given as input and this is indicated by turning the ready signal to high. Next comes the initialization process which involves loading of key and IV into the data bit. This operation is performed by maintaining a count ("init_count") which counts upto 224 which is the number of

bytes needed complete the initialization process are 224. Therefore the initialization process takes 224 clock cycles in total. Once the counter reaches more than 224 then the controller checks for "bdi_ready"and "bdi_ad" signals. If "bdi_ad" is high then the next state is processing associated data if not plaintext/ciphertext is processed. In these two states new counters which counts the number of bytes of associated data ("ad_count") and plaintext/ciphertext ("bdi_count") are initialized. For our design we have taken the length of associated data and plaintext as 128. The controller waits in these state untill the internal state is updated after loading the associated data and plaintext. Both the processes needs 48 bytes to be loaded into the data bit of state update function. Therefore these two states takes 48 clock cycles each. Once the corresponding counter reaches its limit then the controller comes out of that particular state. After Encryption/Decryption is done then the next process is finalization, in which tag generation and validation is done. In order to generate the tag the data bit has to be loaded with 768 0s. So, the controller waits for 96 clock cycles in the tag generation process. This process is handled by using a separate counter "tag_count". The controller waits inside the finalization state until this counter reaches upto 96. After finalization process the controller goes back to the wait state.

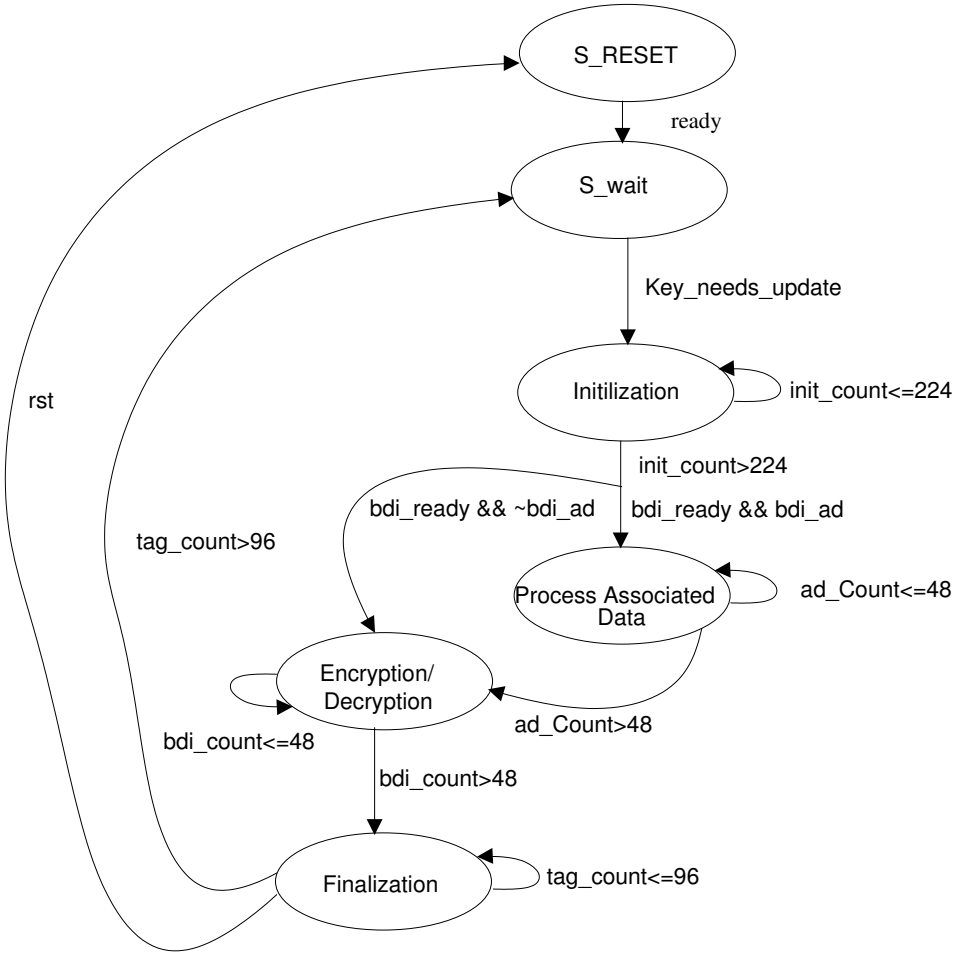


Figure 6.7: Toplevel State Machine of ACORN

Figure 6.8 shows the Toplevel design of ACORN. There are very few signals exchanged between the controller and datapath as compared to the other candidates. The shift signals "sh_key", "sh_iv" and "sh_bdi" are used as the load signals for their respective PISOs. The select signals "sel_a" and "sel_b" are used to select the control bits ca and cb. The enable signal "en_S" is given as input to the register that stores the state. The signals must be enabled as per the operation that needs to be performed. This process is maintained by counters that counts until the process is done. For example, during initialization process the key and nonce must be loaded into the data bit (m_i), and the multiplexer has to select the data input accordingly, for this to happen the select signals and enable signals are maintained until the length of the key (128) and nonce (128) is counted by

using two separate counters for key and nonce. After loading the key and nonce, the process requires loading of a "1" and 15 "0"s into the message bit, this is done by counting the counter until 1535 and setting the select signals and enable signals as needed. Similarly associated data and plaintext are processed by using counters that count upto the length of associated data and plaintext respectively. Inside the controller different states are maintained to process the specific segment of the data.

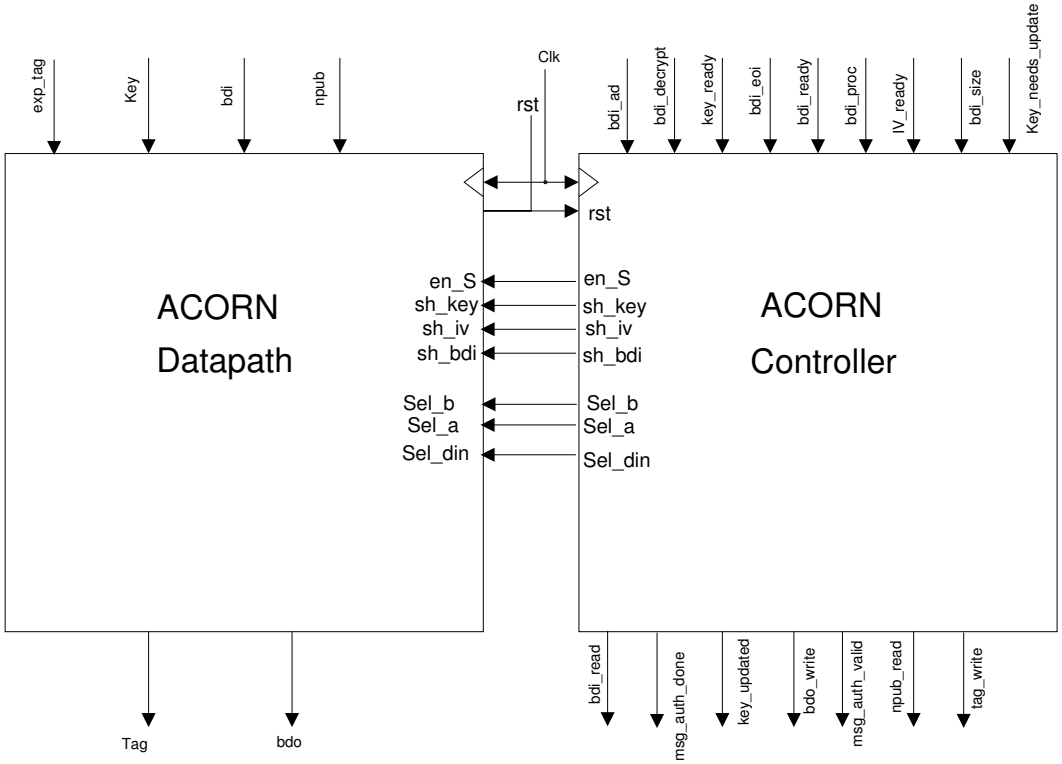


Figure 6.8: Toplevel Design of ACORN

6.4 Lightweight Implementation

6.4.1 Datapath Design

The design is shown in the Figure 6.9.

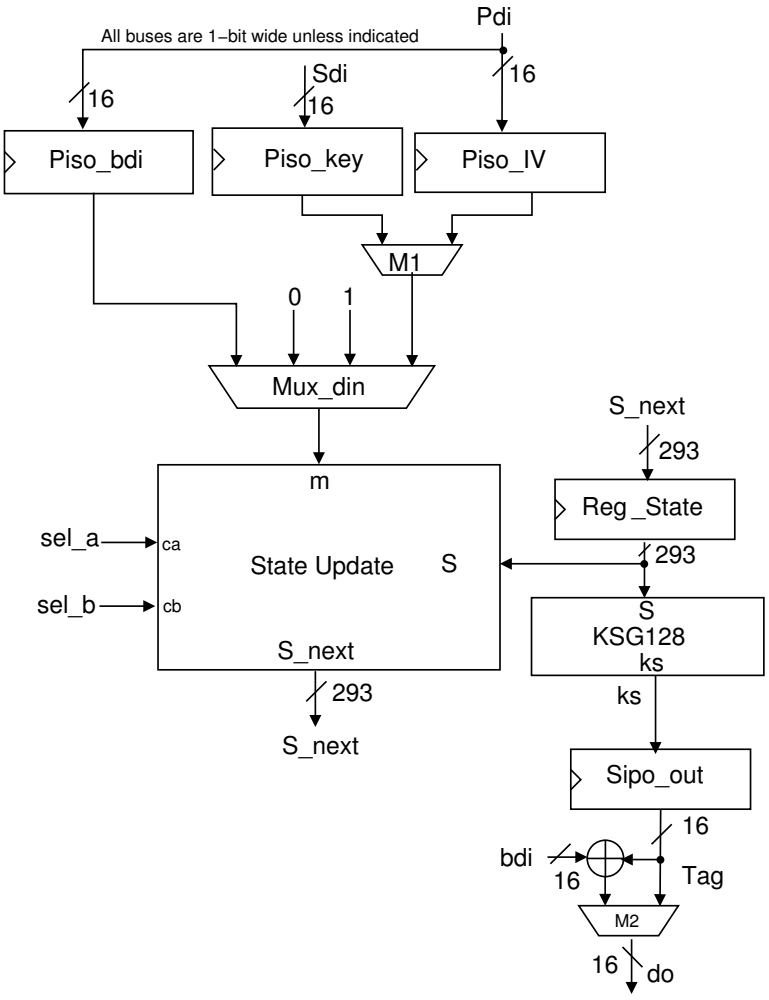


Figure 6.9: Light-Weight Datapath Design of ACORN

The changes made for optimizing the datapath for our lightweight design are listed below:

- The width of the datapath is now changed to a single bit so the PISOs are now designed such that they output single bit at at a time.
- A single stateupdate function and KSG128 function are used instead of eight.
- The size of the muxes is changed to a single bit.

6.4.2 Control Logic Design

Figure 6.10 shows the top level state machine of our lightweight ACORN design. The flow of the controller is similar to that of the fullwidth design.

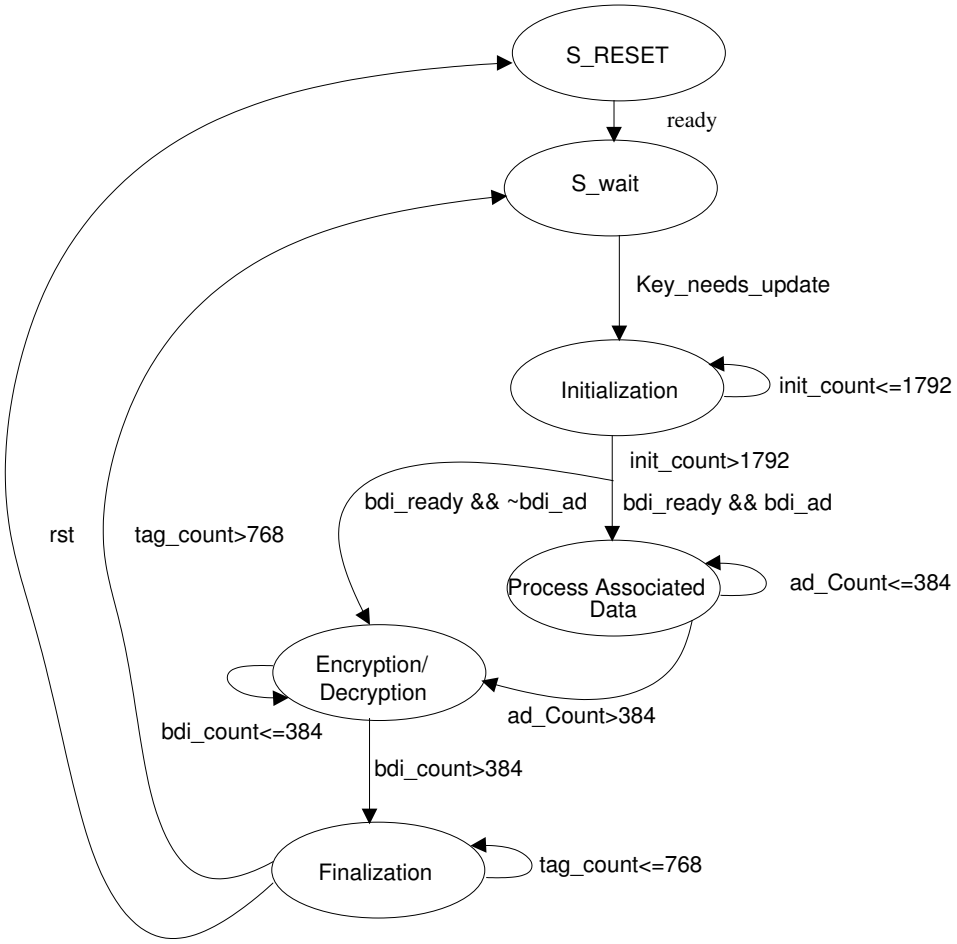


Figure 6.10: Toplevel State Machine of our ACORN Lightweight Design

The controller of the optimized design is designed in the same way as of the fullwidth except that the counters are longer. Hence it takes more clock cycles as compared to the fullwidth design. as we load one bit at a time. The counters are 8 times bigger when compared to those of the fullwidth design. Figure 6.11 shows the top level architecture of the lightweight design of ACORN. The toplevel structure of lightweight design remains same as that of the fullwidth design.

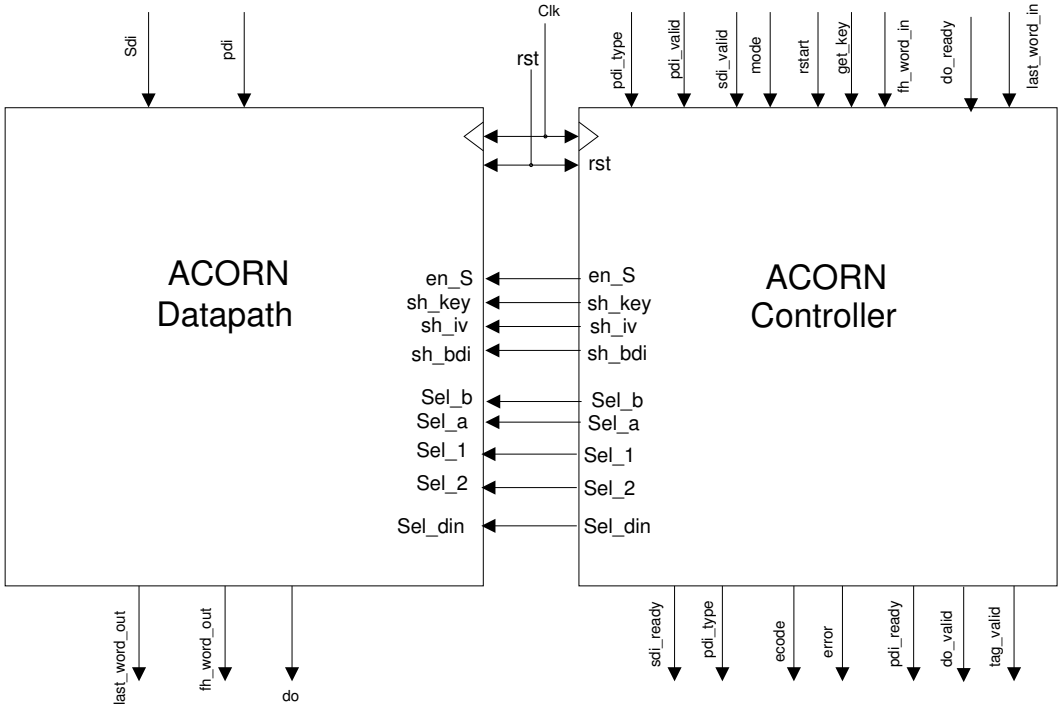


Figure 6.11: Toplevel Structure of ACORN our Lightweight Design

Chapter 7: Performance Evaluation

The results for hardware implementation of selected authentication techniques are tabulated in this chapter.Area utilization and timing results are provided for FPGA implementations.

7.1 Our Implementation Results

7.1.1 Throughput Computations

The parameters that will be used for throughput computations are tabulated in table7.1

Table 7.1: Notations

Symbol	Comment
lm_1	Loading first block message
lm	Loading subsequent blocks of message
lad_1	Loading fist block of associated data
lad	Loading subsequent blocks of associated data
lk	Loading key
ln	Loading nonce or initialization vector
ik	Initialization of key
i	Initialization (does not involve key or iv)
in	Initialization of nonce or iv
pm	Processing message
pad	Processing associated data
z	Finalization
oc	Output block of ciphertext
ot	Output tag

The resultant formula for the number of clock cycles for encrypting N message blocks and authenticating A associated data blocks.

$$clk\_cycles\ (N,A) = i + ik + in + lm_1 + lad_1 + lm\cdot(\text{N-1}) + lad\cdot(\text{A-1}) + pm\cdot(\text{N}) + pad\cdot(\text{A}) + z + oc\cdot(\text{N}) + ot$$

The formula can be simplified as below

$$clk\_cycles\ (N,\ A) = i+ik+in+lm_1+lad_1-lm-lad+ot+z+(lm+oc+pm)\cdot(N)+(lad+pad)\cdot(A) \quad (7.1)$$

Throughput is defined as the number of bits processed per unit of time. We can derive the formula for throughput of encryption/decryption for long messages from (7.1). For very long messages, we can neglect the initialization part so the resultant formula is :

$$Throughput_{ENC/DEC} = \frac{b}{(clk\_cycles(N+1,0) - clk\_cycles(N,0)) \cdot T} Bits/sec. \quad (7.2)$$

where, b is the block size of the algorithm and T is the clock period In the similar way, the throughput of authentication can formulated as below:

$$Throughput_{AUTH} = \frac{b}{(clk\_cycles(0,A+1) - clk\_cycles(0,A)) \cdot T} Bits/sec \quad (7.3)$$

Throughput formulae of our implementations of CAESAR candidates is shown in 7.2

Table 7.2: Throughput formulae for our implementations of CAESAR candidates

Algorithm	Design Type	Block Size (bits) b	Throughput$_{AUTH}$ $\frac{b}{(lad+pad)\cdot T}$	Throughput$_{ENC/DEC}$ $\frac{b}{(lm+pm+oc)\cdot T}$
ACORN	Fullwidth	1	1/ (0+48)· T	1/(0+80+16)· T
ACORN	Lightweight	1	1/ (0+384)· T	1/(0+640+128)· T
SILC	Fullwidth	128	128/ (1+10)· T	128/(1+10+1)· T
SILC	Lightweight	8	8/ (1+10)· T	8/(1+10+1)· T
Joltik	Fullwidth	64	64/ (1+32)· T	64/ (1+32+1)· T
Joltik	Lightweight	4	4/ (1+32)· T	4/ (1+32+1)· T

7.1.2 Resource Utilization

Devices from the Xilinx Spartan-6 and Artix-7 families of FPGAs were targeted for all implementations. The complete resource utilization results of our implementations of all the three candidates are tabulated in 7.3

Table 7.3: Resource Utilization

Candidate	Type	Block Size (Bits)	Slices (Area)		LUTs		FlipFlops		BlockRAMs		Delay (ns)	
			Spartan-6	Artix-7	Spartan-6	Artix-7	Spartan-6	Artix-7	Spartan-6	Artix-7	Spartan-6	Artix-7
ACORN	Fullwidth	1	150	171	402	390	386	336	0	0	2.719	3.677
	Lightweight	1	110	148	286	280	289	289	0	0	4.402	4.098
SILC	Fullwidth	128	529	655	1761	1338	1774	1729	0	0	5.739	4.737
	Lightweight	8	250	274	357	355	166	142	0	0	7.177	5.246
Joltik	Fullwidth	64	861	787	2641	2801	3275	3177	0	0	7.360	5.115
	Lightweight	4	305	354	705	715	802	852	0	0	9.32	7.563

7.1.3 Throughput/Area

Table 7.4 shows the throughput / area calculations of all the implementations.

Table 7.4: Throughput Results of Our Implementations

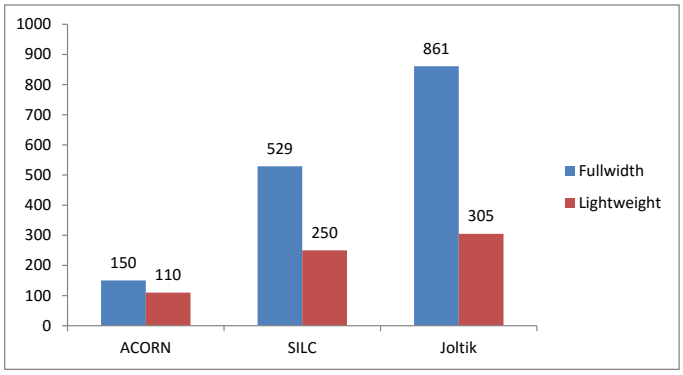
Design Type	Spartan-6								Artix-7							
	Fullwidth				Lightweight				Fullwidth				Lightweight			
Algorithm	Throughput_{AUTH} (Mbps)	TP/Area (Mbps/slice)	$\text{Throughput}_{ENC/DEC}$ (Mbps)	TP/Area (Mbps/slice)	Throughput_{AUTH} (Mbps)	TP/Area (Mbps/slice)	$\text{Throughput}_{ENC/DEC}$ (Mbps)	TP/Area (Mbps/slice)	Throughput_{AUTH} (Mbps)	TP/Area (Mbps/slice)	$\text{Throughput}_{ENC/DEC}$ (Mbps)	TP/Area (Mbps/slice)	Throughput_{AUTH} (Mbps)	TP/Area (Mbps/slice)	$\text{Throughput}_{ENC/DEC}$ (Mbps)	TP/Area (Mbps/slice)
ACORN	7.66	0.05	3.83	0.025	0.5	0.004	0.25	0.002	5.66	0.03	2.55	0.015	0.6	0.004	0.3	0.007
SILC	2,027.5	3.83	1858.6	3.5130	101.3	0.405	92.8	0.371	2456.4	3.75	2251.7	3.43	138.63	0.505	127.36	0.464
Joltik	263	0.3	255.7	0.29	13	0.04	12.6	0.04	350	0.35	340	0.33	16	0.05	15.5	0.04

7.2 Analysis of the Results

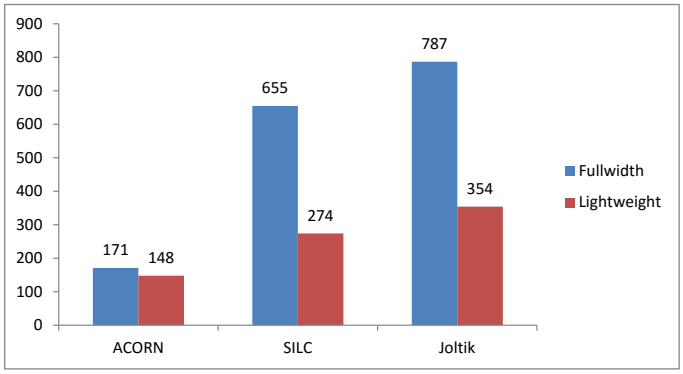
This section covers the analysis of the results obtained.

7.2.1 Area

Of the 3 candidates ACORN uses fewer number of slices as the design is very small when compared to the other candidates. The lightweight version of all the candidates uses fewer number of slices when compared to their respective high speed versions. Figure 7.1 shows the bargraph that compares the number of slices utilized on Spartan6 7.1a and Artix7 7.2b.



(a) Spartan6

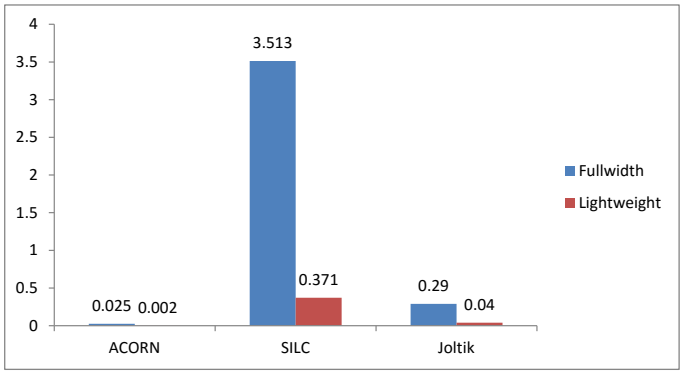


(b) Artix7

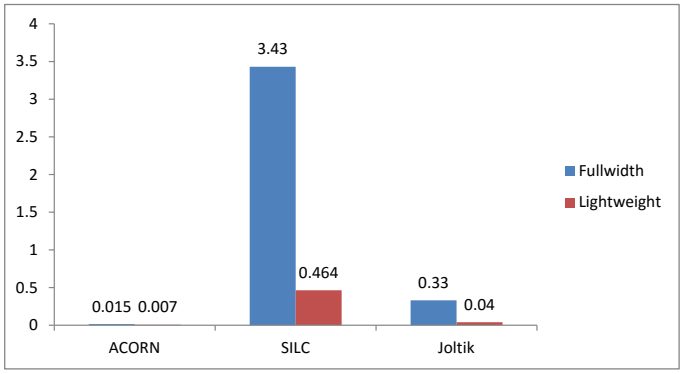
Figure 7.1: Comparison of Slices

7.2.2 Throughput/Area

Of the 3 candidates SILC has the best throughput/area ratio as it uses fewer number of clockcycles and considerable number of slices. Figure 7.1 shows the bargraph that compares throughput/area ratios of Spartan6 7.2a and Artix7 7.2.



(a) Spartan6



(b) Artix7

Figure 7.2: Comparison of Throuhput/Area

7.3 Comparison with Other Published Results

Our implementation results are compared with other candidate implementations.

7.3.1 Fullwidth Designs

The area utilized by our fullwidth implementations is compared with other CAESAR candidate implementation results published in ATHENa results database [1]. These comparisons are for Virtex6 family. The comparisons are listed in table 7.5

Table 7.5: Fullwidth Design Implementation Results Comparison

Candidate	Area [Slices]	Throughput [Mbps]	TP/Area [Mbps/slice]
Related work			
Joltik	582	343	0.590
Ascon	595	3,235	5.437
AES-GCM	999	3,231	3.234
Deoxys	1,165	1,166	1.001
CLOC	1,290	2,709	2.10
OCB	1,418	2,566	1.81
ICEPOLE	2,336	37,480	17.819
Our work			
Joltik	884	453	0.512
ACORN	294	301	0.103
SILC	512	2,779	5.427

Out of all the candidates implemented ICEPOLE has highest throughput and TP/Area as it is permutation based and uses fewer number of clock cycles. Our Joltik implementation used larger number of slices than that of their implementation but the throughput/area ratio is close. ACORN has the lowest throughput of all the candidates that we compared as it is bit wise. ACORN is the only stream cipher based design of all the candidates listed above. Our SILC has considerably good throughput and throughput/area ratio when compared to other implementations.

7.3.2 Lightweight Designs

Our lightweight designs are compared with lightweight design implementation by Pansayya Yalla [52]. These comparisons are for Artix-7 FPGA family.

Table 7.6: Lightweight Design Implementation Results Comparison

Design	Area [Slices]	Throughput [Mbps]	TP/Area [Mbps/slice]
Related work			
AES-GCM	548	630	0.115
Keyak	260	136	1.231
Our work			
Joltik	354	15.5	0.044
ACORN	110	2.55	0.023
SILC	274	13.86	0.05

AES-GCM and Keyak has reasonably better TP/Area when compared our implementations we think the reason for this is that our candidate implementation takes more clock cycles when compare to other lightweight implementations.

Chapter 8: Conclusion

In this final chapter, we summarize the work accomplished and our results obtained and we also state our conclusion about the implemented CAESAR candidates.

8.1 Work Accomplished

Our work first started with characterizing the CAESAR Round 1 submissions under different features. Then we chose 3 candidates that fall into three different categories. We first started with the fullwidth designs of these 3 algorithms and later optimized them. All the 3 algorithms were successfully implemented and verified using post place and route simulation.

8.2 Ranking of Our Implementations

The amount of area used by the 3 candidates can ranked as below:

1. ACORN.
2. SILC.
3. Joltik.

Of all the 3 candidates ACORN uses least area as the size of the datapath is very low when compared to the SILC and Joltik. Between Joltik and SILC, SILC uses less number of resources as the size of the key is less and the design is very simple as it is built on just AES-Enc. Where as Joltik uses a tweakable block cipher as the base and it uses 192 bit tweakey. Processing of key in Joltik-BC takes more area when compared to AES. With respect to throughput/area our implementations can be ranked as below

1. SILC
2. Joltik

3. ACORN

Although ACORN uses least area of all the 3 implementations it has very less throughput as it a stream cipher based algorithm and so all the operations are bitwise. SILC has the highest throughput/area ratio of all the 3 candidates.

Bibliography

Bibliography

[1] ATHENa results database. `http://cryptography.gmu.edu/athenadb/`. Automated Tool for Hardware EvaluatioN project.

[2] CAESAR: Competition for authenticated encryption: Security, applicability, and robustness. `http://competitions.cr.yp.to/caesar-call.html`, March 2014. Call for submissions.

[3] Farzaneh Abed, Scott Fluhrer, John Foley, Christian Forler, Eik List, Stefan Lucks, David McGrew, and Jakob Wenzel. The POET family of on-line authenticated encryption schemes. Submission to CAESAR(Round1), March 2014.

[4] Farzaneh Abed, Christian Forler, and Stefan Lucks. General overview of the first-round caesar candidates for authenticated ecryption. Technical report, Cryptology ePrint report 2014/792, 2014.

[5] Basel Alomair. AVALANCHEv1. Submission to CAESAR(Round1), March 2014.

[6] Elena Andreeva, Begul Bilgin, Andrey Bogdanov, Atul Luykx, Florian Mendel, Bart Mennink, Nicky Mouha, Wang Qingju, and Kan Yasuda.

[7] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. AES-COPA v.1. Submission to CAESAR(Round1), March 2014.

[8] Lear Bahack. Julius: Secure mode of operation for authenticated encryption based on ECB and finite field multiplications. Submission to CAESAR(Round1), March 2014.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission:KETJE v1. Submission to CAESAR(Round1), March 2014.

[10] Bertoni, Guido and Daemen, Joan and Peeters, Michaël and Van Assche, Gilles and Van Keer,Ronny. CAESAR submission:KEYAK v1. Submission to CAESAR(Round1), March 2014.

[11] Alex Biryukov and Dmitry Khovratovich. PAEQ v1. Submission to CAESAR(Round1), March 2014.

[12] Antoon Bosselaers and Fre Vercauteren. YAES v2. Submission to CAESAR(Round1), March 2014.

[13] Mickaël Cazorla, Kevin Marquet, and Marine Minier. Survey and benchmark of lightweight block ciphers for wireless sensor networks. *IDEA*, 64(128):34, 2013.

[14] Avik Chakraborti and Mridul Nandi. TriviA-ck-v1. Submission to CAESAR(Round1), March 2014.

[15] Simon Cogliani, D Stefania Maimut, David Naccache, Rodrigo Portella do Canto, Reza Reyhanitabar, Serge Vaudenay, and Damian Vizar. Offset merkle-damgard OMD version 1.0. Submission to CAESAR(Round1), March 2014.

[16] Nilanjan Datta and Mridul Nandi. ELmDv1.0. Submission to CAESAR(Round1), March 2014.

[17] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. ASCON v1. Submission to CAESAR(Round1), March 2014.

[18] Danilo Gligoroski, Hristina Mihajloska3, Simona Samardjiska, Hakon Jacobsen, Mohamed El-Hadedy, and Rune Erlend Jensen. Pi-cipher v2. Submission to CAESAR(Round1), March 2014.

[19] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, Kerem Varici, François Durvaux, Lubos Gaspar, and Stèphanie Kerckhof. SCREAM and iSCREAM. Submission to CAESAR(Round1), March 2014.

[20] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED block cipher. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 326–341. Springer, 2011.

[21] Sandy Harris. The Enchilada authenticated ciphers, v1. Submission to CAESAR(Round1), March 2014.

[22] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. AEZ v1: Authenticated Encryption by Enciphering. Submission to CAESAR(Round1), March 2014.

[23] Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri, Farnoud Farahmand, Malik Umar Sharif, and Kris Gaj. Gmu hardware api for authenticated ciphers. Cryptology ePrint Archive, Report 2015/669, 2015. `http://eprint.iacr.org/`.

[24] Bart Preneel Hongjun Wu. AEGIS:A Fast Authenticated Encryption. Submission to CAESAR(Round1), March 2014.

[25] Tao Huang Hongjun Wu. JAMBU:Lightweight Authenticated Encryption Mode and AES-JAMBU. Submission to CAESAR(Round1), March 2014.

[26] HOSSEINI, Hossein and KHAZAEI, Shahram. CBA mode (v1-1). Submission to CAESAR(Round1), March 2014.

[27] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, and Sumio Morioka. CLOC: Compact low-overhead CFB. Submission to CAESAR(Round1), March 2014.

[28] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and Eita Kobayashi. SILC: SImple Lightweight CFB. Submission to CAESAR(Round1), March 2014.

[29] Nasour Bagheri Javad Alizadeh, Mohammad Reza Aref. Artemia v1. Submission to CAESAR(Round1), March 2014.

[30] Jeremy Jean, Ivica Nikolic, and Thomas Peyrin. Deoxys v1. Submission to CAESAR(Round1), March 2014.

[31] Jeremy Jean, Ivica Nikolic, and Thomas Peyrin. Joltik v1. Submission to CAESAR(Round1), March 2014.

[32] Jeremy Jean, Ivica Nikolic, and Thomas Peyrin. KIASU v1. Submission to CAESAR(Round1), March 2014.

[33] Jean-Philippe, Aumasson Philipp, and Jovanovic Samuel Neves. NORX. Submission to CAESAR(Round1), March 2014.

[34] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander Christian Rechberger, Peter Schwabe, and Tolga Yalçn. PROSTv1. Submission to CAESAR(Round1), March 2014.

[35] Krovetz, Ted. HS1-SIV(v1). Submission to CAESAR(Round1), March 2014.

[36] Krovetz, Ted and Rogaway, Phillip. OCB (v1). Submission to CAESAR(Round1), March 2014.

[37] Peter Maxwell. Wheesht: an AEAD stream cipher. Submission to CAESAR(Round1), March 2014.

[38] Daniel Penazzi Miguel Montes. AES-CPFB v1. Submission to CAESAR(Round1), March 2015.

[39] Kazuhiko Minematsu. AES-OTR v1. Submission to CAESAR(Round1), March 2014.

[40] PawełMorawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wójcik. Icepole v2. `http://competitions.cr.yp.to/round1/icepolev2.pdf`, Aug 2015. submitted to the CAESAR competition.

[41] Ivica Nikolic. Tiaoxin- 346. Submission to CAESAR(Round1), March 2014.

[42] Daniel Penazzi and Miguel Montes. Silver v.1. Submission to CAESAR(Round1), March 2014.

[43] Francisco Recacha. ++AE v1.0. Submission to CAESAR(Round1), March 2014.

[44] Markku-Juhani O. Saarinen. The STRIBOBr1 Authenticated Encryption Algorithm. Submission to CAESAR(Round1), March 2014.

[45] Yu Sasaki, Yosuke Todo, Kazumaro Aoki, Yusuke Naito, Takeshi Sugawara, Yumiko Murakami, Mitsuru Matsui, and Shoichi Hirose. Minalpher v1. Submission to CAESAR(Round1), March 2014.

[46] Shibutani, Kyoji and Isobe, Takanori and Hiwatari, Harunaga and Mitsuda, Atsushi and Akishita, Toru and Shirai, Taizo. Piccolo: an ultra-lightweight blockcipher. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 342–357. Springer, 201.

[47] Jonathan Trostle. AES-CMCC v1.1. Submission to CAESAR(Round1), March 2014.

[48] Rade Vuckovac. Raviyoyla v1. Submission to CAESAR(Round1), March 2014.

[49] Lei Wang. SHELL v1. Submission to CAESAR(Round1), March 2014.

[50] Hongjun Wu. ACORN: A Lightweight Authenticated Cipher. Submission to CAESAR(Round1), March 2014.

[51] Hongjun Wu and Tao Huang. The authenticated cipher MORUS (v1). Submission to CAESAR(Round1), March 2014.

[52] Panasayya Yalla, Ekawat Homsirikamol, and Jens-Peter Kaps. Comparison of multi-purpose cores of Keccak and AES. In *Design, Automation Test in Europe DATE 2015*, pages 585–588. ACM, Mar 2015.

[53] Bin Zhang, Zhenqing Shi, Chao Xu, Yuan Yao, and Zhenqi Li. Sablier v1. Submission to CAESAR(Round1), March 2014.

[54] Lei Zhang, Wenling Wu, Yanfeng Wang, Shengbao Wu, and Jian Zhang. LAC: A lightweight authenticated encryption cipher. Submission to CAESAR(Round1), March 2014.

[55] Zhang, Liting and Wu, Wenling and Sui, Han and Wang, Peng. iFeed [AES] v1. Submission to CAESAR(Round1), March 2014.

Curriculum Vitae

Upendarreddy Mamidi was born on January 10th, 1992 in Hyderabad,India.He received his Bachelor of Technology degree from Vellore Institute of Technology, Tamilnadu, India in 2013. He started working towards his master's degree in George Mason University from August, 2013. He was involved in teaching various undergraduate courses at George Mason University as a teaching assistant. He is a research student at Cryptographic Engineering Research Group (CERG) with interest in Light weight implementations of Authenticated Encryption schemes.