METHODOLOGY FOR DEVELOPING LIGHTWEIGHT ARCHITECTURES FOR FPGAS

by

Panasayya S.V.V.K Yalla
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Electrical and Computer Engineering

Committee:

_____	Dr. Jens-Peter Kaps, Dissertation Director
_____	Dr. Kris Gaj, Committee Member
_____	Dr. Brian L. Mark, Committee Member
_____	Dr. Robert Simon, Committee Member
_____	Dr. Monson H. Hayes, Department Chair
_____	Dr. Kenneth Ball, Dean, The Volgenau School of Engineering

Date: _____ Fall Semester 2017
George Mason University
Fairfax, VA

Methodology for Developing Lightweight Architectures for FPGAs

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Panasayya S.V.V.K Yalla
Master of Science
George Mason University, 2009
Bachelor of Engineering
Andhra University, 2006

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

Fall Semester 2017
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my mother Anantha Lakshmi and my father Ananda Ramayya. Without their unconditional love and unwavering support, this would not been possible. To my sister Swathi and my brother Satish for their support and encouragement. Last but not least, I dedicate this dissertation to my wife Sharanya for her love and affection.

Acknowledgments

There are many people I must acknowledge who are instrumental in bringing this to a successful completion.

First and foremost, I must thank my advisor Dr. Jens-Peter Kaps. I was fortunate to have him as my mentor. Learnt a great deal of things both academically and personally from him. His constant support, guidance and patience are instrumental in finishing my research work.

Second, I must thank Dr. Kris Gaj for his guidance, support and constructive comments. He always drove us to pay attention to details and aim for perfection. Third, I would like to thank my other committee members Dr. Brian Mark and Dr. Robert Simon for their comments and suggestions.

I would like to thank my friends Rajesh, Mahidhar, Ahmad, Ice, Marcin for their support and advice and making my years at GMU fun and enjoyable. I would also like to thank my other colleagues at CERG group for providing an excellent atmosphere for research. Finally, I would like to thank all my colleagues at Riscure for being very supportive in completing my doctoral studies.

Table of Contents

List of Tables

List of Figures

Abstract

METHODOLOGY FOR DEVELOPING LIGHTWEIGHT ARCHITECTURES FOR FPGAS

Panasayya S.V.V.K Yalla, PhD

George Mason University, 2017

Dissertation Director: Dr. Jens-Peter Kaps

Until now, application specific integrated circuits (ASICs) are the main platform for lightweight cryptography because of their low power consumption and good performance. However, their complex design cycle and very high non-recurring engineering cost limit them to high volume applications. In recent years, low cost and power Field Programmable Gate Arrays (FPGAs) (Xilinx: Spartan-6 and Artix-7; Altera: Cyclone-IV and -V; Actel: IGLOO and ProASIC3) have started emerging, reducing the power consumption gap between ASICs and FPGAs. FPGAs are the ideal platform for fast changing environments and lower volume applications. In spite of these advantages, very little attention has been paid to FPGAs as a target for lightweight cryptography.

Implementing algorithms for lightweight applications is a complex and time consuming task due to interdependencies of the constraints on size, power, energy, and cost. The various design choices such as interface, width of datapath, serialization, pipelining, choice of processing elements etc. determine whether the design meets these constraints. In most cases this results in designs where the datapath width is reduced. However, this is not sufficient, one has to carefully evaluate the trade-off various constraints at every step of the design process. The control unit is an additional hurdle.

Extensive component re-use in the datapath can lead to a very complex control logic which might negate the area savings in the datapath.

In this research, we tackle these problems in three parts. First part involves developing a generalized methodology for making early design choices and various optimizations that can be applied to datapath. The control logic optimization techniques using memories are proposed in the second part. Finally, a tool is developed which optimizes the control logic by using the existing controller or state matrix as the input and transforms it into an optimized controller. This optimized controller is a combination of traditional FSM realized using Flip-flops and combinational logic with fewer states and memories.

Using the proposed methodology, we developed lightweight architectures for block cipher Advanced Encryption Standard (AES) for three different widths, Secure Hash Algorithm-256 (SHA-256), multipurpose AES and Keccak cores, Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) candidates Ketje-Sr, Ascon-128, and Ascon-128a. The effectiveness of the optimization tool is tested using AES128 and Keccak core. We also developed hardware package which supports CAESAR hardware Application Programming Interface (API) for lightweight implementations and evaluated its benefits using Ketje-Sr and Ascon.

Chapter 1: Introduction and Motivation

1.1 Introduction

In the current day of the Internet of Things (IoT) and embedded processor and internet connections are getting integrated seamlessly into our everyday life. Smart-watches, fitness monitors, mobile phones, Global Positioning System (GPS), smart fridges, WiFi thermostats, etc. are a few examples to name. For these devices to provide more customized services, our behavioral patterns, habits, and movements are being tracked. Some of these devices are easy to loose due to their portability and some are prone to malicious hacking as they are constantly connected to internet. Access of this information or control over these devices by unsavory people would lead to undesirable consequences. Hence, safeguarding one's privacy and security is paramount. Cryptographic algorithms are employed to address these issues. Traditional cryptographic algorithms may not be applicable for low end IoT devices as they have limited memory and computational power along with serious power and energy constraints. The branch of cryptography which addresses the security and privacy issues of these devices is called *LightWeight Cryptography (LWC).*

Traditional cryptographic algorithms were made for data processing on a computer. Initially these algorithms were tailored for lightweight cryptographic applications. But this tailoring degraded the performance of the algorithms either in terms of security or speed or both. Therefore, new cryptographic algorithms such as Present [1], Hight [2], XTEA [3] etc. are a few to name were designed exclusively for lightweight cryptographic applications.

In general, LWC algorithms must have the following attributes

- Small internal state→lower area consumption
- Short processing time→lower energy consumption

- Short output→lower communication cost
- Allow serialization→lower power consumption
- Same primitives→same security level as traditional

Some of these attributes are complimentary which can be seen in Figure 1.1. *For example*: Making the algorithm more serialized would reduce area and power consumption at the cost of energy and throughput. Hence, depending on the application and target device, optimization criteria and limitations are determined. In general, corner 1 in Figure 1.1 is the ideal place for lightweight applications. The other two corners 2 and 3 are for high security and low latency applications respectively.

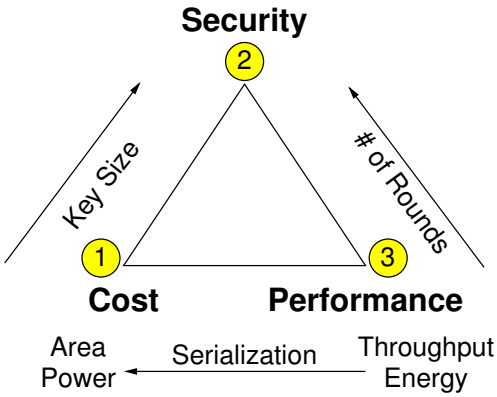

Figure 1.1: Relation of various performance parameters on algorithmic parameter

Based on the target platform, LWC algorithms can be broadly classified into two categories as shown in Figure 1.3. The first category is software oriented algorithms where the algorithms are implemented on a microprocessor (μP) or a microcontroller (μC). Block ciphers XTEA [3] and TWIS [4] are two such software oriented LWC algorithms. The second category is hardware oriented algorithms which are mainly targeted for ASICs and FPGAs. Block ciphers DES [5], HIGHT [2], Present [1] and MIBS [6] are designed for lightweight hardware applications. A generalized classification of various implementation platform is

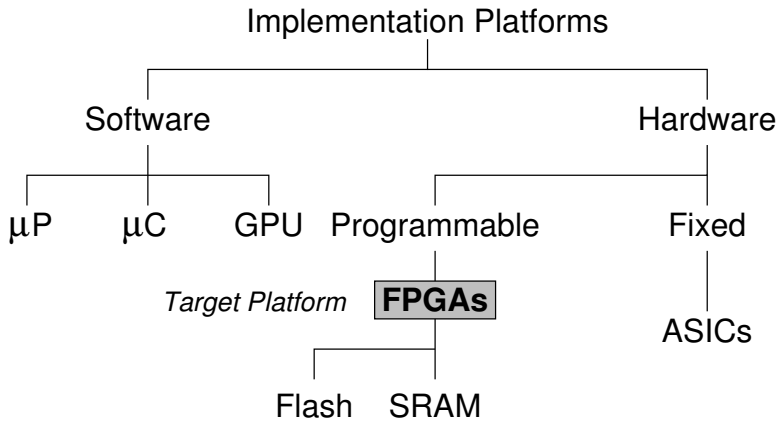

Figure 1.2: Classification of implementation platforms

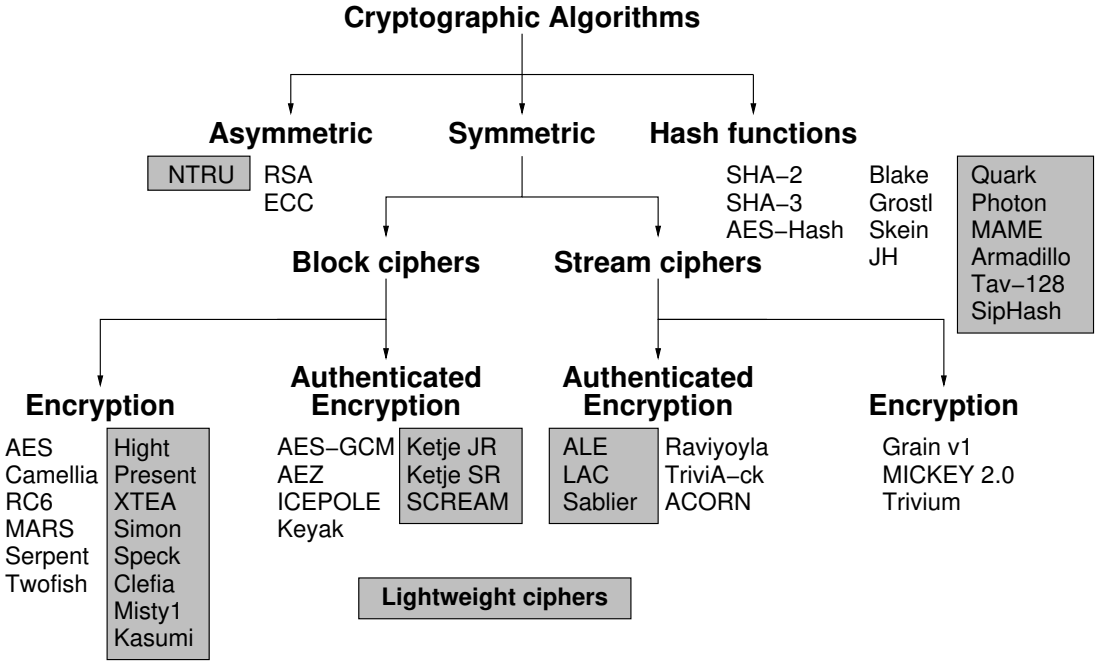

Figure 1.3: Classification of cryptographic algoithms

Table 1.1: FPGA vs ASIC

	FPGA	ASIC
Time-to-market	**Fast**	**Slow**
Upfront NRE cost	**Less**	**High**
Design cycle	**Simple**	**Complex**
Reprogramability	**Yes**	**No**
Unit costs(small volume)	**Low**	**High**
Unit costs(high volume)	**High**	**Low**
Power Consumption	**High**	**Low**
Internal clock speeds	**Low**	**High**
Custom Capability	**No**	**Yes**

shown in Figure 1.2. In our current research, we limit our focus only to FPGAs as the target platform.

1.2 Motivation

Until now, application specification integrated circuits (ASICs) are the primary platform for lightweight cryptography because of their low power consumption and good performance. The Table 1.1 shows the comparison of FPGAs and ASICs in terms of several factors that determine the target platform. The ones marked in **green** are the favorable ones. However, their complex design cycle and very high Non-Recurring Engineering (NRE) cost limit them to high volume applications. In recent years, low cost and power Field Programmable Gate Arrays (FPGAs) (Xilinx: Spartan-6 and Artix-7; Altera: Cyclone-IV and -V; Actel: IGLOO and ProASIC3) have started emerging, reducing the power consumption gap between ASICs and FPGAs. FPGAs are the ideal platform for fast changing environments and lower volume applications. Reconfigurability of FPGAs allows the system to be upgraded if ever the need arises. This upgrade may be critical due to security threats or change in the operating conditions. In spite of these advantages, very little attention has been paid for lightweight cryptography for FPGAs. Hence in this dissertation, we explore these devices for lightweight cryptographic applications.

Using FPGAs as the target platform for lightweight applications is relatively a new area.

Therefore, no set of guidelines exists for developing lightweight architectures. This problem coupled with complexity involved due to interdependencies of constraints on size, power, energy, and cost make it a challenging task. In this dissertation, we tackle these problems.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 covers the background regarding FPGAs and their power consumption and Read-Only Memory (ROM)-based Finite State Machines (FSMs). Previous work is described in chapter 3 in three sections depending on the area of research. In Chapter 4 we outline our contributions in this dissertation. The methodology for developing lightweight architectures and control logic optimization tool is described in Chapter 5. Lightweight implementations of block cipher AES and SHA-256 are described in Chapter 6. We evaluate the CAESAR hardware API for lightweight implementations in Chapter 7. In Chapter 8, we present multi purpose cores of Keccak and AES and compare them. The Chapter 9 describes the lightweight multi-mode AES implementation for ASCIs. Finally, we present our conclusions and future work in Chapter 10.

Chapter 2: Background

This chapter provides the background for FPGAs and their power consumption and ROM-based FSMs.

2.1 FPGA

Field Programmable Gate Arrays (FPGAs) are ICs which can be configured using HDL in the field after manufacturing. These FPGAs contains a matrix of programmable logic blocks and a mesh of programmable interconnects connecting them. In general, a logic block contains the following:

1. Look Up Tables (LUTs)
2. Flip-flops (FFs)
3. Miscellaneous logic

LUTs can be configured to perform complex combinational functions. FFs are memory units within the logic block which can either be used purely as storage units or can be used to realize sequential logic in combination with LUTs. Apart from LUTs and FFs, logic blocks also contain miscellaneous logic elements such as multiplexers, logic gates, carry chain etc. which increases the versatility of logic block. In addition to logic blocks and interconnects, FPGAs also contain

1. I/O blocks: I/O pins which can be programmed as inputs or outputs
2. Memory blocks: Large memory blocks which allows for on-chip memory.
3. Digital Signal Processing (DSP) blocks: Dedicated blocks for performing addition/subtraction and multiplications

4. Clock distribution network: Network of dedicated lines for routing clock.

5. Phase-Locked Loops (PLLs): Serves as frequency synthesizer for a wide range of frequencies.

There are multiple FPGA vendors in the current market. However, only three vendors Xilinx, Altera, and Microsemi together comprise 90% of market share in 2012 [7]. Hence, FPGAs from these three vendors are considered in this dissertation.

2.2 Power Consumption in FPGAs

The power consumption in an FPGAs is dependent on its underlying technology. Even though there exists many FPGAs based on various technologies, we limit our discussion only to namely SRAM (Static Random Access Memory)-based (Xilinx and Altera) and flash-based (Microsemi) FPGAs. Flash-based FPGAs consume less power than SRAM-based FPGAs [8]. For these two FPGAs, the following are the major components of power consumption

1. Static power: Power consumed by the device when it is turned on but not actively performing any operation also called standby power.

2. Dynamic power: Power consumed by the device when it is actively performing operations.

3. Inrush power: It is the power consumed by the FPGA during power-up

4. Configuration power: Power consumed during the configuration of the FPGA upon power up. This power is specific to SRAM-based FPGAs due to their non-volatile nature.

5. Sleep-mode power: Power consumed by an FPGA when it is in sleep mode or low power mode.

In this dissertation, we limit our focus only to power consumed during normal operation of an FPGA or execution power ($P_{execution}$) which is the sum of static power ($P_{stateic}$) and dynamic power ($P_{dynamic}$) as shown in Eq 2.1.

$$P_{execution} = P_{static} + P_{dynamic} \tag{2.1}$$

The static power ($P_{stateic}$) depends on several factors such as transistor leakage, temperature, and supply voltage. On the other hand, dynamic power ($P_{dynamic}$) depends on switching frequency (f), supply voltage (V) and load capacitance (C) and switching activity (α) as shown in Eq 2.2 .

$$P_{dynamic} = \alpha \cdot C \cdot V^2 \cdot f \tag{2.2}$$

The major contributors for $P_{dynamic}$ are power consumption due to interconnects, logic, clocking, and I/O blocks. Within these four, power consumption due to interconnects is found to be the dominant one [9] and [10].

2.3 ROM-based FSMs

Finite State Machines (FSMs) or *Finite State Automaton* provide the mathematical abstraction for the design of digital systems and computer programs. These FSMs are one of the four major families of automaton, the other three are Pushdown, Linear-bounded and Turing machines. FSMs consists of a set of states with a start state, inputs, outputs, and a transition function which maps current inputs and state to a next state. Depending on relation of the current input (X) on the output (Y), FSMs can be classified into two categories called *Mealy* [11] and *Moore* [12] machines named after G.H. Mealy and E.F. Moore in recognition of their work. In a Mealy machine, the output depends on both current input and the state where as in Moore machine it only depends on the current state as shown in Figures 2.2 and 2.1 respectively.

For both Mealy and Moore machines, there are three components namely next-state

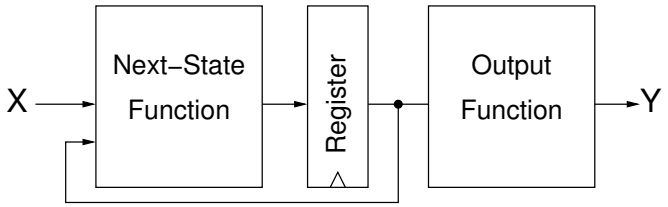

Figure 2.1: Moore machine

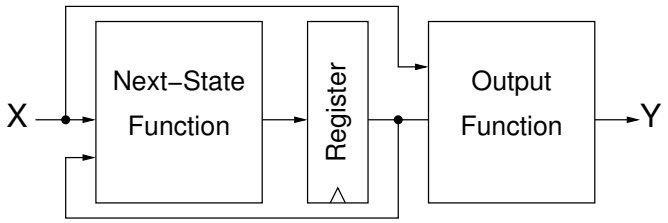

Figure 2.2: Moore machine

function, output function and state register. The traditional way of realizing FSM in digital circuits is by using combinational logic for next-state and output functions and Flip-Flops (FFs) for state register. Another way to synthesize FSM is by using memories (RAM/ROMs) which are found to be more efficient in terms of power, speed, and resource usage [13–16]. The idea of using ROMs for FSM is not a novel one as it is similar to microprogram control unit in a processor.

Notations:

Number of input variables	: M
Number of output variables	: N
Number of FSM states	: P
Input variable set	: X=$\{x_1, x_2, \cdots, x_M\}$
Output variable set	: Y=$\{y_1, y_2, \cdots, y_N\}$
State variable set	: S=$\{s_1, s_2, \cdots, s_P\}$
Number of bits needed to represent state variable S (binary encoding)	: R=$\lceil \log_2 P \rceil$

To convert a traditional FSM into a ROM-FSM, all control signals for each operation in a given clock cycle or state are combined to form one control word. Apart from control signals, the control word also contains the state bits (bits need to represent a state) as shown in Figure 2.3. Such control words for each state are stored in a memory which can be accessed by an address. The addresses for the memory can be generated by using a simple counter or a shift-register. The size of the memory for the FSM is determined by

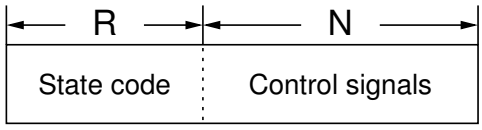

Figure 2.3: Control word

number of inputs, outputs, and states as given by the equation 2.5. If no optimizations are considered, the size of the memory would be very large. Hence, when using ROM-based FSMs, various optimization techniques are used to reduce its size. A simple ROM-FSM is shown in Figure 2.4.

$$\text{Number of bits in each control word} = R + N \tag{2.3}$$

$$\text{Number of memory address bits} = R + M \tag{2.4}$$

$$\text{Total size of the memory} = 2^{R+M} \cdot (R + N) \tag{2.5}$$

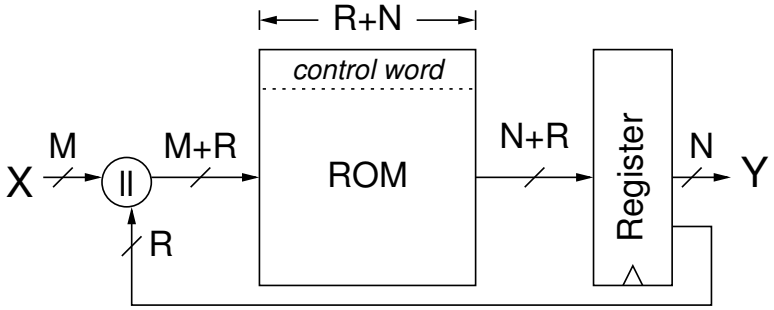

Figure 2.4: A simple FSM based on memory (ROM)

Chapter 3: Previous Work

In this chapter we summarize the most significant work in the area of lightweight architectures. This chapter is broken down into three individual sections for ease of understanding and describing.

3.1 Survey of Lightweight Algorithm Implementations

In [17] and [18], a brief survey of lightweight algorithms implementation on ASICs which includes both symmetric and asymmetric cryptographic algorithms along with hash function are presented. Additionally in [17], the effect of various algorithmic factors such as structure, functional primitives and storage requirements on energy consumption are analyzed. Furthermore, ASIC implementations of various lightweight block ciphers are analyzed with a particular focus on low-latency in [19] and energy in [20].

Various lightweight FPGA implementations for block ciphers AES, Camellia, XTEA, Present, Hight are reported in [21–26]. All these implementations are targeted for Xilinx Spartan-3 FPGA. The widths of their datapath range from 8-bit to 32-bit and consume an area of 100 to 400 slices with some using dedicated memory blocks. In [27], FPGA implementations of stream ciphers are presented. Lightweight FPGA implementations of SHA-3 competition round-2 candidates in [28] and round-2 candidates in [29,30]. In [28,29] similar optimizations are applied on all candidates with same design constraints to make a fair comparison. The effect of unrolling of various lightweight block cipher implementations on energy consumption is investigated in [31]

3.2 Optimization Techniques for Datapath

The most straightforward approach for reducing the area of the datapath is folding. Vertical folding reduces the datapath width while horizontal folding reduces the size of processing elements while maintaining the datapath width. How many times and in which direction a design can be folded depends on the algorithm. The extent to which folding can be applied to the SHA-3 candidates and how much it affects their throughput and throughput over area ratio has been examined in [32].

Another technique is reusing of processing elements. The benefit of applying this technique and additionally with vertical folding at multiple levels down to single processing elements, not just the datapath as a whole is shown in [32]. Both folding and reuse of processing elements minimize the area consumption at the cost of an increased number of clock cycles. In some of the algorithms like SHA-2, straight forward application of pipelining techniques is not trivial due to data dependency. So an optimization technique called quasi-pipelining is applied to reduce the critical time. These technique rearranges order of operations along with introduction of pipeline registers which optimizes the critical path along with reduction in latency [28] and [29].

Several optimization techniques which use the inherent features such as LUT based memories (DRAMs) and shift registers (SRL16) of Xilinx Spartan-3 FPGAs([33]) are proposed in [21, 22] and their effectiveness is demonstrated by implementing the block ciphers Camellia [34], Present [1], and Hight [2]. Furthermore, SRL16 are found to be well suited for optimizing serialized implementation of block cipher LED and PHOTON [35].

For optimizing the leakage power in an FPGA, a technique called *sleep mode* where the unused logic blocks and flip-flops are put to sleep is proposed in [36, 37]. In [38], this technique is further improved and applied to the embedded memories. In order to optimize the dynamic power consumption, several techniques which include use of 5-LUTs, new routing algorithms, clustering schemes, double-edge-triggered flip-flops are proposed in [39–41]. An Overview of various power optimizations that can be applied to FPGAs are

presented in [42].

3.3 Optimization of ROM-based FSMs

Using memory units in an FPGA for implementing FSM was first proposed in [14] along with optimization technique called *fuzzy state encoding technique* to reduce its size. This optimization technique can only be applied to FSM which contain *fuzzy bits* in their state codes. In [15,16], ROM-based FSMs are implemented using embedded memory blocks which are found to be more power saving. Using functional decomposition where the number of variables in a function is reduced is proposed in [43] for optimizing logic function implemented on an FPGA. This technique is further explored for reducing the size of ROM-based FSMs in [13]. With the use of *"don't cares"* in logic functions, the size of the ROM-based FSM is reduced in [16].

3.4 Summary

There are no generalized optimization techniques which can applied to basic building blocks while considering various interdependent design constraints. Furthermore, transforming a traditional FSM into ROM-based FSM and applying optimization techniques is complex and ime consuming. Having a tool to perform this task would be very helpful. But no such tool exists which can transform a traditional FSM described in VHDL as input and generates an optimized ROM-based FSM in VHDL.

Chapter 4: Contributions

The main goal fo this dissertation is to reduce the complexity and time required in designing efficient lightweight architectures targeted for FPGAs. We address this in two parts. In first part, we proposed a generalized methodology for developing architectures targeted for lightweight applications. For the second part, we optimize the control logic with a tool we developed using python. This tool optimizes the controller using memories. We evaluated this tool using block cipher AES and Keccak Core as the test cases.

Using the proposed methodology and optimization techniques, we designed lightweight architectures of AES128 with 8, 16, 32-bit datapaths. We also developed lightweight architectures of CAESAR candidates KETJE-SR, ASCON-128, and ASCON-128A and used them for evaluating CAESAR hardware LWAPI package. Additionally, we investigate the benefits of building cryptographic services based on the same cryptographic primitive for lightweight applications using AES and Keccak. Furthermore, we developed a lightweight 8-bit AES core targeted ASICs which supports both encryption and decryption, multiple modes, and two key sizes.

Chapter 5: Methodology for Developing Lightweight Architectures

In this chapter, we put forth our methodology for developing lightweight architectures. Designing a lightweight architecture is a complex task compared to high speed implementation due to various limiting factors like size, power, energy, and cost. Usually digital designs, can be broken down into three parts namely datapath, controller and top-level interface as illustrated in Figure 5.1.

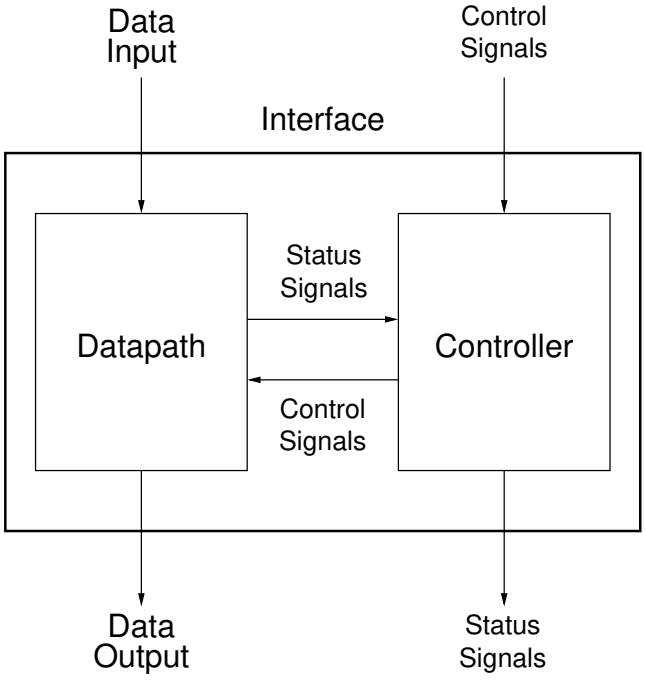

Figure 5.1: Top-level block diagram of an architecture

Datapath is a unit where the data is processed and controller is the one which controls

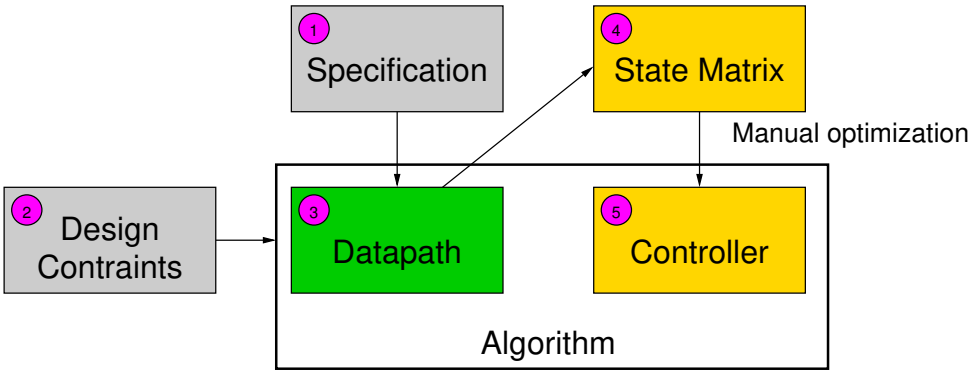

Figure 5.2: Lightweight architecture design flow

other units by providing timing and control signals. The top-level interface encompasses both datapath and controller providing an interface between these two units and the outside world.

Developing lightweight architectures is not a straight forward approach given the inter-dependency of controller and datapath. Reducing the datapath width and reuse of resources may not be ideal if the control logic negates any of the saving achieved. The Figure 5.2 shows generalized design flow we used in development of lightweight architectures.

We break down various optimization techniques that can be applied in developing lightweight architectures into three categories.

1. Top-level optimizations: We identify the factors that influence both datapath and control unit and analyze their effect on the performance in this section (1 and 2 in Figure 5.2).

2. Datapath optimizations: In this section, we analyze various existing techniques and also develop new ones for datapath optimizations (3 in Figure 5.2).

3. Controller optimizations: Several techniques for optimizing the control logic using ROM-based FSMs are explored in this section (4 and 5 in Figure 5.2).

Table 5.1: Comparison of interface widths with respect to lightweight applications

Interface	Advantages	Disadvantages
Serialized	• Least number of I/Os required	• More time spent on communication •Additional resources for converting *serial to word-size* for input and *word-size to serial* for output
Word-size/ *N*.word-size	• Small number of I/Os required • Usually optimum for performance	• Small penalty on communication
Parallelized	• Less time on communication	• Large number of I/Os required • May not be able to leverage memories for input and output storage

5.1 Top-level Optimizations

5.1.1 Interface

The top-level interface which determine the number of Input-Output pins (I/Os) needed is dependent on the application. Usually, the number of I/Os must be limited to small number. Having a large number of I/Os would require larger FPGAs which increases the cost. On the other hand, keeping it very low would require more time spent on communication which degrades the performance. Hence a proper balance must be struck between the two.

The other effect of the interface is due to width of the I/O bus. If the width is smaller or not a multiple of word size of the algorithm, additional storage units may be needed as seen in [28, 29]. Table 5.1 lists the comparison of interface widths.

5.1.2 Width of datapath

Due to various constraints such as area, power, energy, etc. an algorithm cannot be implemented at full width (each round operation takes one clock cylce) for lightweight applications. So an algorithm is folded both horizontally (width of datapath) and vertically

Table 5.2: Optimum datapath widths for some of the cryptographic functions

Algorithm	Width(bits)	Function
AES128	32	Mixcolumns
Camellia	8	8x8 Sbox
HIGHT	8	8-bit Addition
Present	16	Permutation
SHA-3	64	64-bit rotation
Ketje-Jr	8	8-bit rotation
Ketje-Sr	16	16-bit rotation
Ketje-Minor	32	32-bit rotation
Ketje-Major	64	64-bit rotation
Ascon-128	64	Linear diffusion
Ascon-128a	64	Linear diffusion

(number of rounds processed in one clock-cycle). The width of the functions in the algorithm determines the optimum width of the datapath called as natural width. Having datapath width smaller than the natural width would incur a penalty in terms of area and performance. For example, the optimum width of datapath for AES[44] is 32-bit due to its Mixcolumns function. If a full 128-bit width datapath of AES requires 1 clock-cycle for each round, a 32-bit datapath would need 4 clock-cycle (128/32=4). But in case of 8 or 16-bit datapaths, it would require more than 8 and 16 clock-cycle (128/16=8, 128/8=16) respectively as they are below the natural width. These additional clock-cycles would also increase the complexity of the controller i.e. large counters, more states etc. In some cases, additional storage may be required.Some of the optimum datapath widths for some of the cryptographic functions is shown in Table 5.2

5.1.3 Choice of an FPGA

The underlying features of an FPGA varies depending on the vendor, family and type of packaging. Due to these variations, the performance of an algorithm implementation varies across FPGAs. Even on the same device but using different resources of an FPGA would end with different results [29]. Hence choosing an appropriate FPGA is essential for efficient implementation. Table 5.3 lists all the currently available FPGAs from the three

major vendors Xilinx, Altera and Microsemi along with their major features.

5.2 Datapath Optimizations

Datapath consists of storage (key, plaintext, ciphertext, etc) and processing elements (eg. round function). The available resources in a given FPGA and characteristics of the function determine the efficient way to implement them. In lightweight architectures, storage elements account for majority of area. These storage elements can be implemented in an FPGAs in the following ways

1. Flip-flops
2. LUT based memory
3. Dedicated Memory
4. LUT based shiftregisters (only in Xilinx FPGAs)

Flip-flops

Flip-flops(FF) are one way to implement storage elements. Each of FPGA programmable logic blocks consists of not just Flip-flops but also other resources such as LUTs, multiplexer. Using flip-flops only as a register would render these resources unusable. Also each of programmable block consists of limited number of flip-flops and therefore require a large number of blocks. For example, in Xilinx 6 and 7 series FPGAs, each SLICE consists of 8 flip-flops. If we have to storage 128-bits, we would require 16 SLICEs (128/8). Furthermore, additional resources such as multiplexers are required to select bits for processing and output given that only part of the state is processed in each clock cycle due to scaling of design. The advantage of using a flip-flops is that all of the data stored is readily accessible. Hence these can be used for temporary storage small data blocks.

Table 5.3: List of FPGAs currently available from the three major vendors

Vendor	Family	Technology (nm)	Cost	Power	Speed	# of LUT inputs	Dedicated Memories	Multiplier size
Xilinx	Spartan-6	45	low	low	low	6	18Kb	18x18
	Artix-7	28	lowest	lowest	low	6	18Kb	25x18
	Kintex-7	28	moderate	moderate	low	6	18Kb, 36kb	25x18
	Virtex-7	28	high	high	high	6	18Kb, 36kb	25x18
Altera	Cyclone-IV	60/65	lowest	lowest	low	4	9Kb	18x18
	Cyclone-V	28	lowest	lowest	low	8	10Kb	18x18
	Arria -V	28	low	moderate	moderate	8	20/10Kb	18x18
	Stratix-IV	40	high	high	high	6	9Kb, 144kb	9x9 12x12 18x18 36x36
	Stratix-V	28	high	high	high	6	9Kb, 144Kb	9x9 12x12 18x18 36x36
	Stratix-10	14	high	low	high	4	576b, 4Kb, 576Kb	9x9 18x18 36x36
Microsemi	IGLOO/e	130	low	low	low	3	4Kb	
	IGLOO nano	130	low	lowest	low	3	4Kb	
	IGLOO Plus	130	low	low	low	3	4Kb	
	ProASIC3/e	130	low	low	low	3	4Kb	
	ProASIC3 nano	130	low	low	low	3	4Kb	
	ProASIC3 Plus	130	low	low	low	3	4Kb	
	IGLOO 2	low	low	low	low	4	18Kb	18x18
	SmartFusion 2	65	low	low	low	4	18Kb	18x18

LUT Based Memories

Some of the programmable logic blocks LUTs can be configured as memories (MLAB in Altera and SLICEM in Xilinx). These are localized memories which can be cascaded to realize deeper memories with minimal penalty on timing. The resources required for these memories depend on memory depth and number of output bits required. These memories can be configured as synchronous/asynchronous read and synchronous write. For example, in Xilinx 6 and 7 series FPGAs, for a state of 256-bits at 8-bit wide words, we would require only 2 slices(32x8 memory) which is far less as compared to 32 slices if flip-flops are used. In [21], it is observed that the area consumption for key and data storage for Camellia [34] block cipher is reduced by a factor of 4.

Dedicated Memories

As shown in Table 5.3, all of FPGAs have dedicated memories with varying sizes and features. These dedicated memories offer a large amount of memory space for storage and can be configured in various configurations single-, dual-port, and quad port memories. The limit on the number of I/O lines limits the number of independent values and number of bits that can be accessed in a single clock. This may lead to use of additional resources to improve performance. Additionally using these resources might cause a degradation in performance [45].

LUT Based Shift-Registers

LUT bases shift registers additional features of Xilinx FPGAs. Each of the 6-input LUTs of SLICEM in Xilinx 6 and 7 series FPGAs can be configured as 32-bit shift register(SRL32). The number of slices required for implementing a shift register depends on the number of bits to be stored and the number of taps. Taps are are positions of a shift register where data can be read from or written too. The tap breaks the chain and places a Flip-flop. The Figure 5.3 shown an example of 32-bit shift register with taps at 12, 11, 10, and 0. The advantage of using a shiftregister is that it does not require any addressing which is

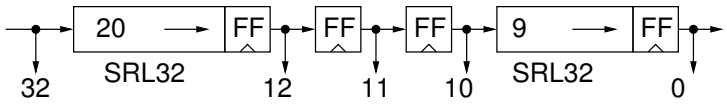

Figure 5.3: 32-bit shiftregister using SRL32s in Xilinx 6 and 7 series FPGAs

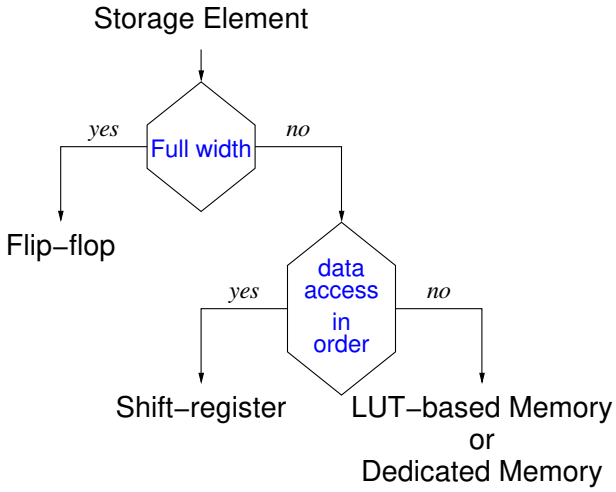

Figure 5.4: Choosing storage element implementation option

required in case of 5.2 and 5.2. The Figure 5.4 shows the decision tree for choosing the option for storage element implementation.

The Figure5.5 shows architecture of AES state using non memory element ie. FFs. If the same state is implemented using memories, it requires 60% less resources. (Table 5.4).

Table 5.4: Comparison of realizing AES state using flip-flops and LUT based Memory on a Xilinx Aritix-7 FPGA in terms of FFS, LUTs, and slices

	FFs	LUTs	Slices
FFs	64	72	21
DRAM	8	8	8

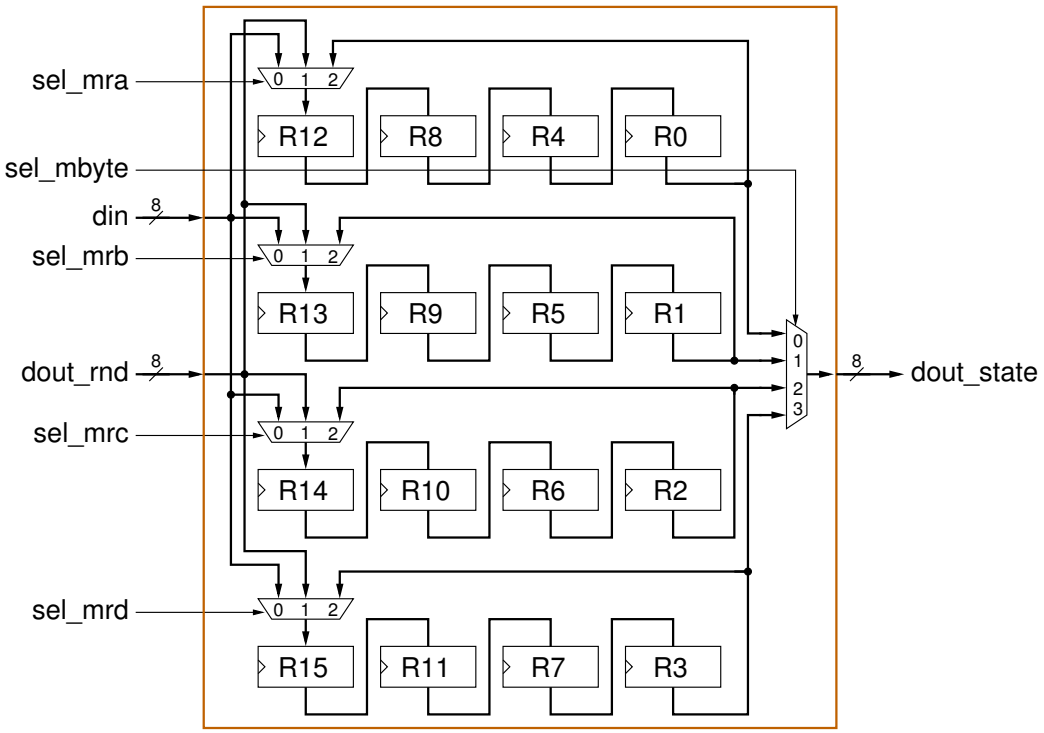

Figure 5.5: State of Mulit-Mode AES using flip-flops

5.3 Control Logic Optimizations

Once the datapath is designed, a state table (matrix with list of control signal in columns and states in rows)is developed. An snippet of one such state table is shown in Figure 5.6. The main goal here is to transform state table into a ROM-based FSM. But given the dependency of some of the control signals on inputs, transforming into a pure ROM-FSM is not feasible. Therefore, we choose a hybrid approach consisting of an FSM with small number of states for input dependent control signals and ROM/s for all other signals. The control signals separation is performed by dividing the whole state table into three phases.

1. Input phase: Loading of inputs
2. Iterative phase: round operations
3. Output phase: writing the result out

Control Signals during round operations

	Main Counter				Round Counter									KEY		Data		SR3				Datapath							Address			
Tcll	R	En	L	O/P	R#	R	En	Rclk	y4	y3	y2	y1	yo	R	L	wr	Add	en	wr			MD1	MD2	MD3	MD4	MD5	MD6	MD7	MA1	MA2	MA3	MA4
32	0	0	0	2	1	0	1	0	0	0	0	0	0	x	x	0	0	1	1	x	x	1	0	0	1	0	0	0	1	x	0	0
33	0	0	0	2	1	0	1	1	0	0	0	0	1	x	x	0	5	1	1	0	1	1	0	0	1	0	0	0	1	x	0	0
34	0	0	0	2	1	0	1	2	0	0	0	1	0	x	x	0	10	1	1	0	1	1	0	0	1	0	0	0	1	x	0	0
35	0	0	0	2	1	0	1	3	0	0	0	1	1	x	x	0	15	1	1	0	1	1	0	0	1	0	0	0	1	x	0	0
36	0	0	0	2	1	0	1	4	0	0	1	0	0	13	0	1	0	0	0	0	1	1	1	0	1	0	1	0	1	x	0	0
37	0	0	0	2	1	0	1	5	0	0	1	0	1	14	1	1	5	0	0	0	1	1	1	0	1	0	0	0	1	x	0	0
38	0	0	0	2	1	0	1	6	0	0	1	1	0	15	2	1	10	0	0	0	1	1	1	0	1	0	0	0	1	x	0	0
39	0	0	0	2	1	0	1	7	0	0	1	1	1	12	3	1	15	0	0	0	1	1	1	0	1	0	0	0	1	x	0	0
40	0	0	0	2	1	0	1	8	0	1	0	0	0	x	x	0	4	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
41	0	0	0	2	1	0	1	9	0	1	0	0	1	x	x	0	9	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
42	0	0	0	2	1	0	1	10	0	1	0	1	0	x	x	0	14	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
43	0	0	0	2	1	0	1	11	0	1	0	1	1	x	x	0	3	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
44	0	0	0	2	1	0	1	12	0	1	1	0	0	0	4	1	4	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
45	0	0	0	2	1	0	1	13	0	1	1	0	1	1	5	1	9	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
46	0	0	0	2	1	0	1	14	0	1	1	1	0	2	6	1	14	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
47	0	0	0	2	1	0	1	15	0	1	1	1	1	3	7	1	3	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
48	0	0	0	2	1	0	1	16	1	0	0	0	0	x	x	0	8	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
49	0	0	0	2	1	0	1	17	1	0	0	0	1	x	x	0	13	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
50	0	0	0	2	1	0	1	18	1	0	0	1	0	x	x	0	2	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
51	0	0	0	2	1	0	1	19	1	0	0	1	1	x	x	0	7	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
52	0	0	0	2	1	0	1	20	1	0	1	0	0	4	8	1	8	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
53	0	0	0	2	1	0	1	21	1	0	1	0	1	5	9	1	13	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
54	0	0	0	2	1	0	1	22	1	0	1	1	0	6	10	1	2	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
55	0	0	0	2	1	0	1	23	1	0	1	1	1	7	11	1	7	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
56	0	0	0	2	1	0	1	24	1	1	0	0	0	x	x	0	12	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
57	0	0	0	2	1	0	1	25	1	1	0	0	1	x	x	0	1	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
58	0	0	0	2	1	0	1	26	1	1	0	1	0	x	x	0	6	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
59	0	0	0	2	1	0	1	27	1	1	0	1	1	x	x	0	11	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
60	0	0	0	2	1	0	1	28	1	1	1	0	0	8	12	1	12	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
61	0	0	0	2	1	0	1	29	1	1	1	0	1	9	13	1	1	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
62	0	0	0	2	1	0	1	30	1	1	1	1	0	10	14	1	6	0	0	0	1	1	0	1	0	0	0	0	1	x	0	0
63	0	0	0	2	1	0	1	31	1	1	1	1	1	11	15	1	11	1	0	0	2	1	0	1	0	0	0	0	1	x	0	0

Figure 5.6: Snippet of AES128 8-bit datapath state table

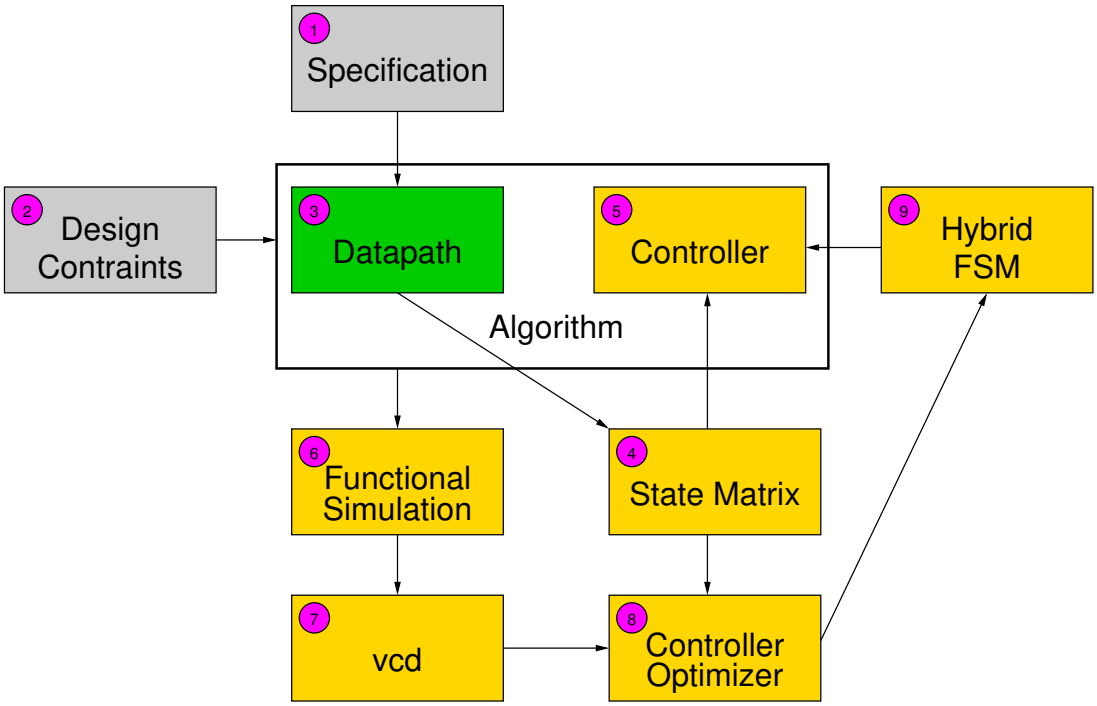

Figure 5.7: Design flow with controller optimization

Most of the control signals such as enables for counters, registers, memory write etc are involved in the input and output phase end in FSM while the iterative phase in ROM. Manually translating into a hybrid FSM is complex and tendencies process. Hence, we developed a tool based on python to transform the state table into a hybrid FSM (5 to 9 in Figure 5.7).

5.3.1 General Control Logic Optimization Strategy for Tool

Let us assume that state table in Figure 5.9 which contains 19 states. The **blue** columns denoted A are two identical columns and **brown** not of A . The rows within **green** rectangles (B1, B2, and B3) are identical except the ones within **magenta** (C) and **red** (D) rectangles.

For this state table, the tool first applies the *vertical optimization* i.e. the number of the control signals are reduced by removing the redundant ones. In the current example shown in Figure 5.9, there are two identical columns denoted by A. One of the two A blocks is

removed during vertical optimization.

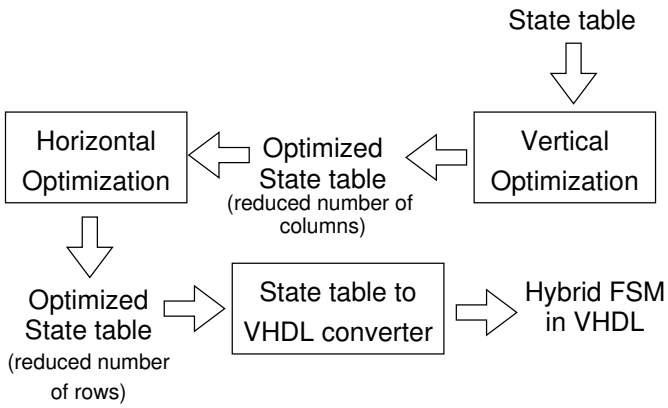

Figure 5.8: FSM optimization flow

Horizontal optimization techniques are applied in the next step which reduces the number of states. For the given examples, the blocks C and D are merged and transformed into a ROM while each of the blocks B1, B2, B3 are reduced to a single state. The optimized state table now contains 13 states which can be seen in Figure 5.10. In the final step, we manually transforms the optimized state table back into VHDL. The tool combines the blocks **magenta** (C) and **red** (D) and generates a ROM in VHDL.

The end result is a hybrid FSM as it contains both traditional and ROM-based FSMs is shown in Figure 5.11. In addition to these FSMs, the hybrid FSM might have a couple of counters or shift-register to generate addresses for ROMs and multiplexers to choose between the main and ROM-based FSM control signals. During the horizontal and vertical optimizations, we incorporated the existing techniques proposed in [13–16, 43].

The address for the ROMs are generated either by using a counter or a shift-register. Counters are based on adder logic whose delay increases with its size where as shift-register do not suffer from this draw back. Since both are realized using look-up tables and flip-flops, there may not much difference for smaller sizes.

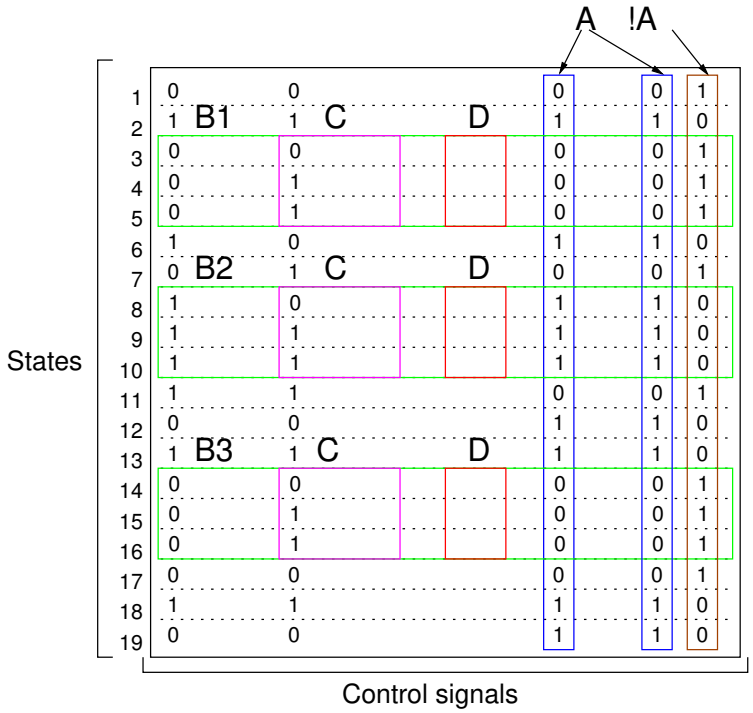

Figure 5.9: State table

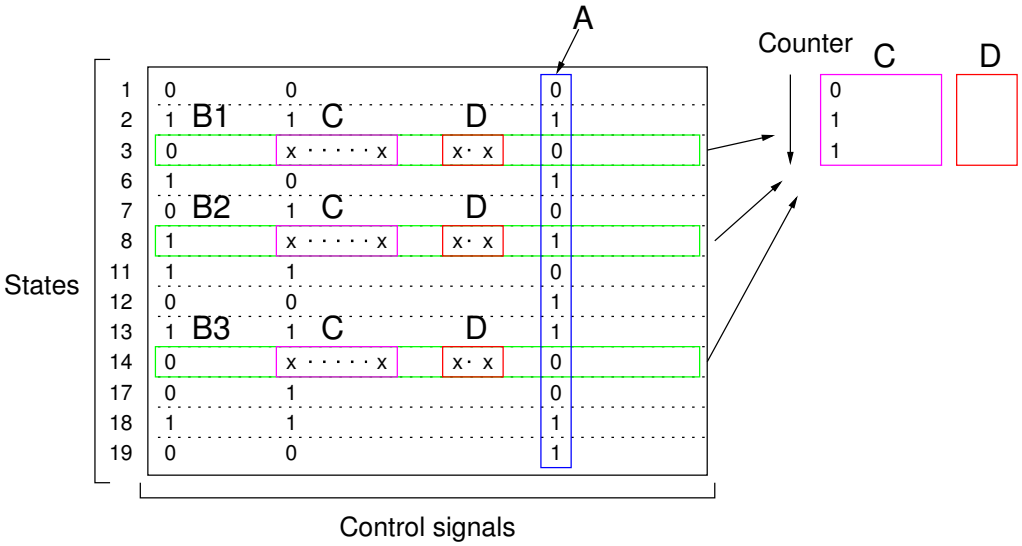

Figure 5.10: Optimized state table

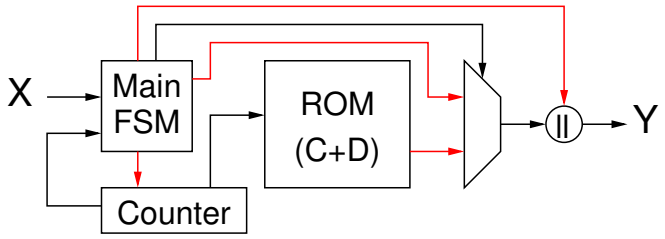

Figure 5.11: Hybrid FSM

Table 5.5: Comparison of controller for AES128 6.1.4 using traditional approach vs tool optimized on Xilinx Aritix-7 FPGA

Controller	States	Control signals	FFs	LUTs	Slices	Frequency(MHz)
Traditional	20	56	9	55	33	202.38
Tool optimized	8	48	11	40	25	157.60

5.3.2 Optimization Test Case

In order to test the effectiveness of the control logic optimization tool, we consider following two test cases.

- **CAES:1** AES128 lightweight architecture (described in Section 6.1.4)
- **CAES:2** Lightweight Keccak core (described in Section 8.3)

5.3.3 CASE:1

In this case, we considered the AES128 implementation with 16-bit datapath. For this datapath, we manually developed the state table using spreadsheet. This state table is then translated into a traditional FSM i.e using FFs and combinational logic. The same state table is fed as the input to the tool and obtained an optimized version of controller. The tool was able to reduce the number of control signals from 56 to 48 and states from 20 to 8. The tool generated a 10x12-bit ROM which contains the control signals for the round operations. There is a slight dip in the obtained maximum frequency for tool optimized version. We believe this may be due to addition of multiplexer at the end of ROM output. This clearly shows the benefit of using the developed tool for optimizing control logic.

5.3.4 CASE:2

In this case, the optimized controller is obtained by feeding the state table developed for the Keccak datapath described in Section 8.3. The tool was able to find three groups (16x34, 16x50, 16x64)within the state table which were moved into the memory. We obtain memories with higher depths (34, 50, and 64) as each round in Keccak 8.3 takes 163 clock cycles. Additionally, the number of control signals were reduced from 68 to 48.

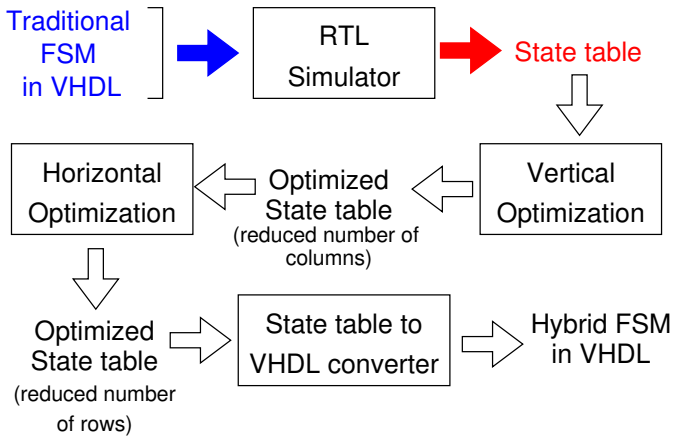

Figure 5.12: Generation of state table using RTL simulator

5.3.5 Generation of State Table Using Simulator

Value Change Dump (VCD) is one the two formates a toggle data is generated by Electronic Design automation(EDA) simulation tools. The toggle data is used as input in power estimation tools such as Xilinx Xpower. We use this toggle data to generate the state tables for existing designs (Figure 5.12). If the design adheres to the format as shown in Figure 5.1 where datapath and controller are separate modules, just adding the controller module would capture toggle data of all control signals. Otherwise, one need to add all this signals individually.

Generation of VCD

We capture the toggle data by using either Xilinx ISIM or modelsim. The commands specified below are the ones used for generating the VCD dumpfile. The first command sets the filename, followed by the variables to capture and module name. Once these are set, the simulation needs to run for at least one whole operation of the cryptographic function. The last command flushes all the data into a specified file.

```
$vcd dumpfile <filename.vcd>
$vcd dumpvars -m <module>
$run <run time>
$vcd dumpflush
```

5.3.6 Translation of VCD to State Table

We developed a tool in python which reads the VCD file and generates the state table. Since the VCD captures only the toggle data i.e, control signals that are unchanged are not captured. Therefore, we need to fill in this missing data. We do this by saving previous values and update only the changed value. After populating the data, we filter out the rows where clk value is 1. The resulting matrix now becomes the state table for the design. The obtained state table can now be given as input to control logic optimization tool to generate the optimized controller as illustrated in Figure 5.7.

Chapter 6: Lightweight Implementations of AES128 and SHA-256

In this chapter we describe various lightweight architectures developed by using the optimizations described in the Chapter 5. All of these designs are targeted for Xilinx FPGAs.

6.1 Lightweight AES Architectures

6.1.1 Interface

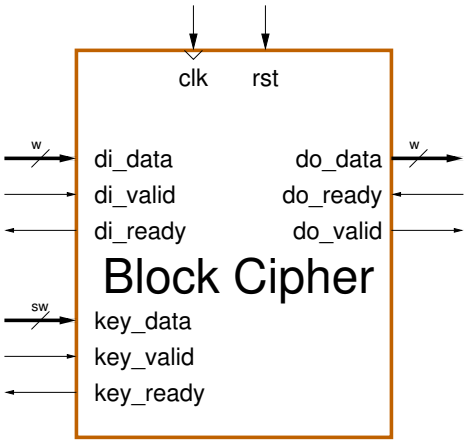

Figure 6.1: Top-level interface *(W, SW = 8, 16, and 32).*

The block cipher architectures for lightweight use the interface shown in Figure 6.1.1. and description of the ports are found in Table 6.1. Another design constraint considered for these implementation is that the cipher core should be able to support I/O feedback

modes i.e output is xored with input as illustrated in Figure 6.1.1. In order accommodate this design constraint, both input and output has to be synchronised.

Table 6.1: CipherCore Port Descriptions.

Name	Direction	Size	Description
Data Input & Output			
key_data	in	SW	Key data
di_data	in	W	Block data input
do_data	out	W	Block data output
Key Control			
key_valid	in	1	Key data is valid
key_ready	out	1	CipherCore is ready to receive a new key
Input Data Control			
di_valid	in	1	BDI data is valid
di_ready	out	1	CipherCore is ready to receive data
Output Data Control			
do_valid	out	1	DO data is valid
do_ready	in	1	ready to receive data.

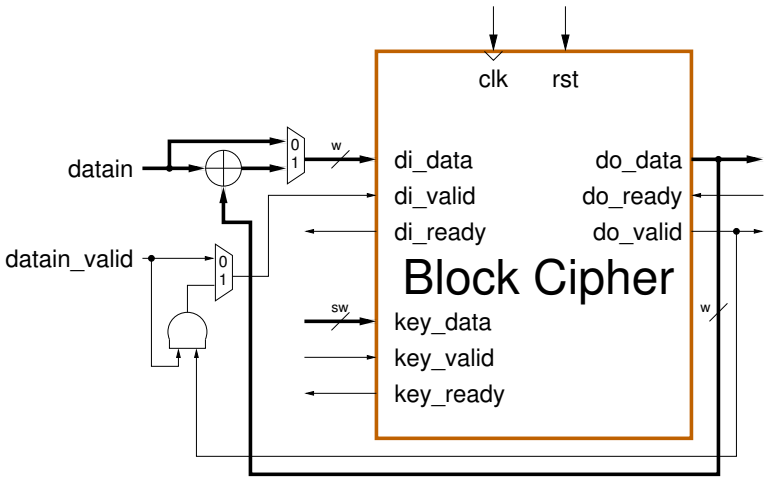

Figure 6.2: Top-level interface with feedback*(W, SW = 8, 16, and 32).*

6.1.2 AES Algorithm

Advanced Encryption Standard (AES) [44] is a 128-bit block cipher which supports key lengths of 128, 192, and 256-bits. For lightweight applications, we consider 128-bit key length to be adequate. Therefore, we consider a 128-bit key size for all our implementations. The round function of AES consists of four functions namely SubBytes, ShiftRows, MixColumns and AddRoundKey. For each of 10 round operations, the AES state is transformed using these four functions except the last round where MixColumns are skipped.

6.1.3 Lightweight Architecture with 8-bit datapath

In this architecture, the reduction in area is achieved by scaling the datapath width to 8-bit. This is well below the natural width of AES (32-bit). Hence, we require additional resources for storing intermediate values (four 8-bit registers A3 to A0) as shown in Figure 6.1.3 and also pay the penalty in higher latency. The state is stored in a single port RAM (DATA DRAM) and key in two DRAMS (K0 DRAM and KR DRAM) one for original key and the other for round key. We assume that the key is rarely changed in lightweight applications. Hence, preserving the original key would save additional clock cycles for loading the key for each block of data. The single port RAM of state both for read and write operations would incur penalty as simultaneous read and write from two different memory locations is not feasible. We over come this by storing the round output to the location we are reading leading to misaligned data for the subsequent round. Therefore, a different sequence of addresses are required for each round operation. We noticed that the misalignment of data is off by a fixed offset between any two subsequent rounds. For each round, this offset is corrected and a new sequence of addresses are generated using two shiftregisters (SR1 and SR2) and a 4-bit adder and added to Therefore, no additional latency is incurred. For key scheduling, the addresses are generated by using a counter (KC), subtracter and a ROM. The round constant used in key scheduling are generated using a shift registers(SR3). The SubBytes module is shared between round and key scheduling to keep the area foot print low. Overall, it takes 310 clock cycles for encrypting a single 128-bit block of data.

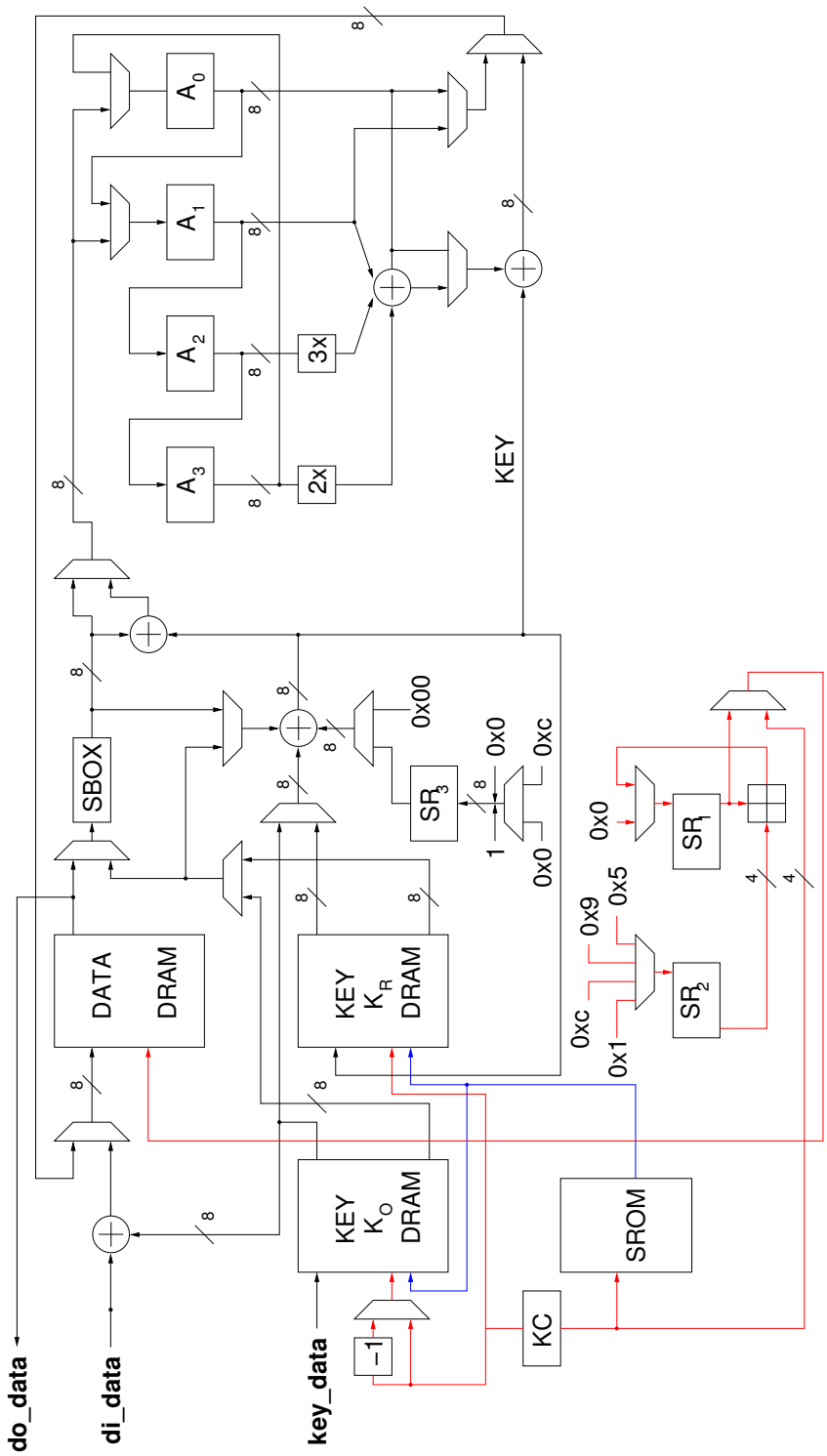

Figure 6.3: 8-bit lightweight architecture of AES128

6.1.4 Lightweight Architecture with 16-bit datapath

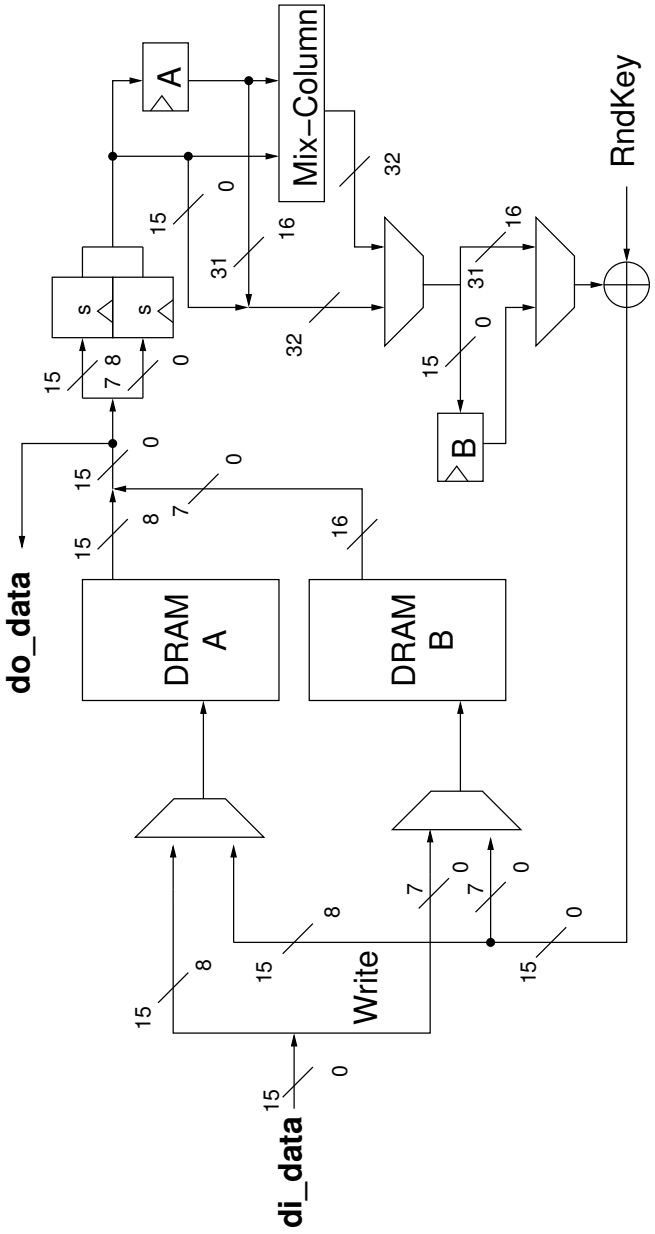

Figure 6.4: 16-bit lightweight architecture of AES128

A 16-bit datapath is considered for this architecture which is also below the natural

width. Therefore, we require additional resources (register in Figure 6.1.4) for storing the intermediate round outputs. As AES round function requires byte addressable words, using a single 16-bit memory is not a preferred choice. Therefore, two memories (DRAM A and DRAM B) are used for storing the state with each storing a byte of the 16-bit word. An additional register (B) is used to store the lower 16-bits of MiXColumns function output. Using this additional register, the latency is improved by a factor of two. Furthermore, the complexity of control logic is reduced due to decreased number of clock cycles for each round and fewer control signals due to increased regularity.

The key is stored in a single dual port RAM where lower address locations are used for storing the original key and higher addresses for round keys. The round keys are generated on the fly. This architecture requires 88 clock cycles for encrypting a single 128-bit block of data.

6.1.5 Lightweight Architecture with 32-bit datapath

The 32-bit datapath of lightweight AES is shown in Figure 6.1.5. As the datapath width is equal to the natural width, no additional resources are required for intermediate values. As stated above, due to byte addressable word constraint of AES round function, a single 32-bit wide memories is not the optimum choice to store the state. Therefore, four 8-bit wide memories are used for storing the four bytes of a 32-bit word. The key is stored in a dual-port ram and the round constants in a ROM. Separate SubBytes modules (S) for round and key scheduling are used as sharing would incur 20% penalty on the latency.

6.1.6 Implementation Results

We implemented our designs on a Xilinx Spartan-6, Virtex-6, Artix-7, and Virtex-7 FPGAs and the results are shown in Table 6.2. Here we only compare with lightweight implementation of AES128 from the literature. All our archiectures achieve a higher throughput with less resource usage as compared to [31] which is a full width implementation. In case of our 32-bit architecture, TP/A ratio is almost twice that of [31]. Due to high latency,

Table 6.2: Comparision of our lightweight implementation of block ciphers with previous results (TP= Throughput; TP/A= Throughput/Area; TW = This Work)

Device	Cipher	Throughput (Mbps) at f_{max}	Area (Slices)	Throughput/ Area (Mbps/Slices)
Spartan-6	AES-8[TW]	59.47	86	0.87
	AES-16[TW]	236.66	133	1.78
	AES-32[TW]	489.58	170	2.88
	AES[31]	213.00	668	0.32
Virtex-6	AES-8[TW]	98.22	71	1.38
	AES-16[TW]	383.58	115	3.34
	AES-32[TW]	834.75	212	3.94
Artix-7	AES-8[TW]	71.92	64	1.12
	AES-16[TW]	280.26	125	2.24
	AES-32[TW]	639.64	219	2.92
Virtex-7	AES-8[TW]	102.69	62	1.66
	AES-16[TW]	467.25	120	3.89
	AES-32[TW]	829.27	190	4.36
Zynq	AES-8[TW]	71.41	67	1.07
	AES-16[TW]	291.84	134	2.18
	AES-32[TW]	548.47	190	2.89

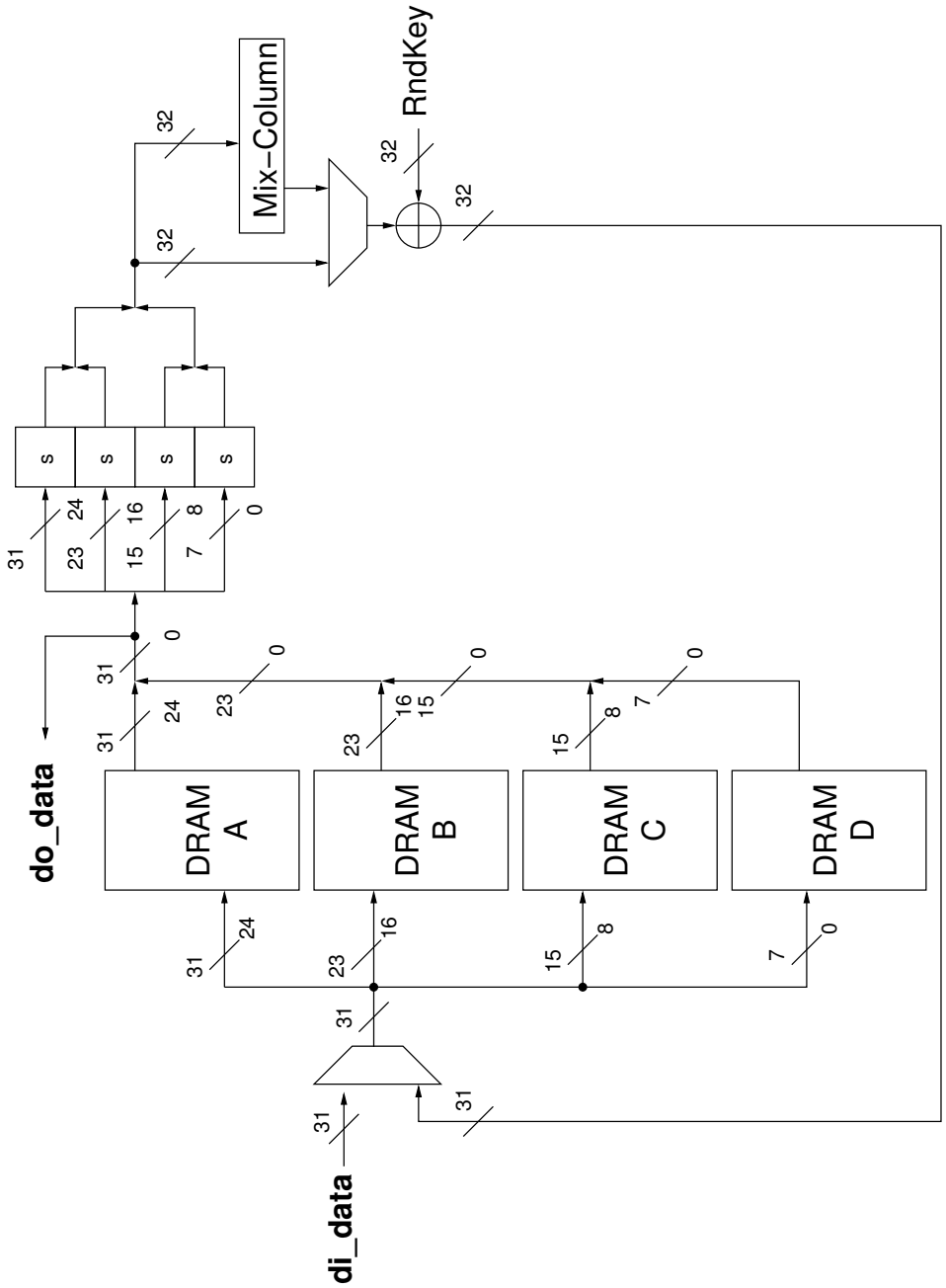

Figure 6.5: 32-bit lightweight architecture of AES128

8-bit architecture achieves a throughput of 59.47 Mbps which is 4.0 and 8.2 times lower as compared to 16-bit and 32-bit architectures.

Table 6.3: Results for our AES implementation compared to Other Block Ciphers and the eSTREAM Portfolio Ciphers on Xilinx FPGA(TW = This Work)

Design	Maximum Delay (ns)	Clock Cycles per block	Block Size (bits)	Key Size (bits)	Area (slices)	Block RAMs	Throughput (Mbps) at f_{max}	Throughput/ Area (Mbps/slice)	Device
AES-8 [TW]	13.48	310	128	128	226	0	30.64	0.07	xc3s500e-5
AES-16[TW]	3.14	88	128	128	590	0	131.68	0.12	xc3s500e-5
AES-32[TW]	1.51	44	128	128	595	0	274.05	0.24	xc3s500e-5
Present [22]	8.78	256	64	128	117	0	28.46	0.24	xc3s50-5
HIGHT [22]	6.12	160	64	128	91	0	65.48	0.72	xc3s50-5
Camellia [21]	7.95	875	128	128	318	0	18.41	0.06	xc3s50-5
AES [23]	14.21	534	128	128	393	0	16.86	0.04	xc3s50-5
AES 8-bit [24]	14.93	3900	128	128	124	2	2.2	0.01	xc2s15-6
AES [25]	20.00	46	128	128	222	3	139	0.27	xc2s30-5
TinyXTEA-3 [26]	15.97	112	64	128	254	0	35.78	0.14	xc3s50-5
Grain v1 [27]	5.10	1	1	80	44	0	196	4.45	xc3s50-5
Grain 128 [27]	5.10	1	1	128	50	0	196	3.92	xc3s50-5
MICKEY v2 [27]	4.29	1	1	80	115	0	233	2.03	xc3s50-5
MICKEY 128 [27]	4.48	1	1	128	176	0	223	1.27	xc3s50-5
Trivium [27]	4.17	1	1	80	50	0	240	4.80	xc3s50-5
Trivium (x64) [27]	4.74	1	64	80	344	0	13,504	39.26	xc3s400-5

For comparing with other lightweight implementation of block ciphers and eSTREAM ciphers, we implemented our three AES architectures on Xilinx Spartan-3 FPGA. A larger device is chosen as there are not enough I/O ports for our AES-32 architecture. Considering block ciphers, our three AES architectures equal or outperforms all other implementations with respect to throughput. The only exception is AES-8 architecture which has about 47% throughput of HIGHT [22]. All stream ciphers outperform all block cipher implementations when we consider throughput/area ratio. This may be due to their 80-bit key size except in case of Grain 128 and MICKEY 128 where 128-bit key is considered. Overall AES-32 has the highest throughput and HIGHT [22] for throughput/area ratio for block ciphers. One interesting aspect noticed is that the 16-bit AES architecture consumes as much area as AES 32-bit architecture. But, both throughput and throughput/area are reduced by about

50%. This may be due to the additional cost associated for datapath width being less than 32-bit which is the natural width of AES.

6.2 Lightweight SHA-256 Architecture

As part of SHA-3 competition, we developed lightweight architecture of SHA-2 to evaluate other candidates. With Xilinx Spartan-3 FPGA as the target device an area budget of 700 slices in case of logic only (No use of BRAMs) and 500 slices with use of 1 BRAM are considered as the design constraints.

6.2.1 Interface

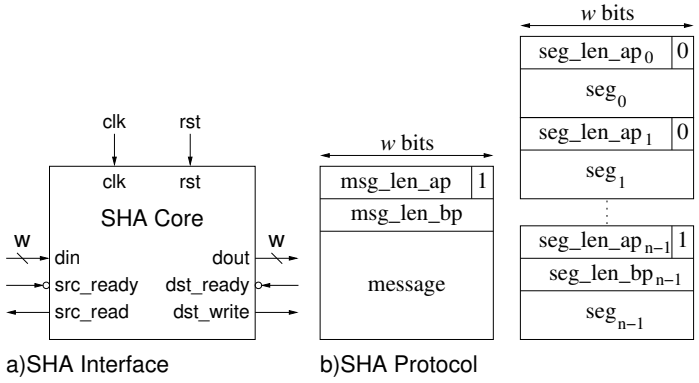

Figure 6.6: Interface and protocol for our SHA cores

We based our hardware interface and I/O protocol (Fig. 6.6) on the one presented in [46] and updated in [47]. The SHA Core assumes that its inputs and outputs are connected to FIFOs. We believe that the FIFO interface model proposed in [46] is very suitable for lightweight implementations. In its simplest form a FIFO is a single w-bit wide register with minimal logic to support the handshake of read/write and ready. This can easily be interfaced to a micro-controller or other circuitry in an embedded system. Lightweight applications usually have smaller databus sizes than the 32 or 64 bits proposed in [46].

Therefore, we use a databus width w of 16 bits. The protocol supports two scenarios: 1) when the message length is known and 2) when the message length is not known. In case 1) the message is sent as a single segment starting with the message length after padding "msg_len_ap" in 32-bit words concatenated with a '1' followed by the message length before padding "msg_len_bp" in bits followed by the message. The "msg_len_bp" is needed by several algorithms even when the message is already padded. In case 2) the message can be processed in segments $seg_0, seg_1, \cdots, seg_{n-1}$. Each segment $seg_0, \cdots, seg_{n-2}$ is headed by the segment length after padding "seg_len_ap" concatenated with a '0' followed by the segment of the message. The last segment seg_{n-1} follows the format of case 1). It contains a block of the message and must contain all padding. The formulae to compute the total number of bits before padding and after padding are:

$$msg\_len\_ap = \sum_{i=0}^{n-1} seg\_len\_ap_i \cdot 32$$

$$msg\_len\_bp = \sum_{i=0}^{n-2} seg\_len\_ap_i \cdot 32 + seg\_len\_bp_{n-1}$$

Furthermore in order to conserve logic resources needed for message counters, we limit the total amount of data in a single message to 2^{32} bits i.e. 4 Gbits which we believe is sufficient for lightweight applications.

6.2.2 SHA-256 Algorithm

The SHA-256 uses six logical functions Ch, Maj, $\Sigma 0$, $Sigma1$, $sigma0$, and $sigma1$. Each of these functions operates on 32-bit words resulting in a new 32-bit words. These six functions are used in one of the three processing steps. The first processing step is message expansion where a 512-bit message block is expanded into 2048-bit message using two functions $\sigma 0$ and $\sigma 1$. The second processing step is round operation which uses eight working variables $a, b, .., h$. These eight working variables are initialized with initial hash values and updated

using 2048-bit message, sixty four 32-bit round constants and four functions $\Sigma 0$, $\Sigma 1$, Ch and Maj. The final step is intermediate hash generation where the eight working variables are added with initial hash values to obtain new intermediate hash values.

6.2.3 Lightweight SHA-256 Architecture

We implemented two versions of each algorithm, one which utilizes only logic resources (Logic version) and one that additionally utilizes a single Block RAM (BRAM version).

Using BRAMs

Our implementation of SHA-256 with BRAM uses it in Dual-port mode to store message, working variables, round constants, initial and final hash values (6.7). The datapath is quasi-pipelined to reduce the critical time and clock cycles. Most of the pipeline registers except R1, R2 and R3 does not cost any additional area. The initialization of the working variables and intermediate hash values is performed while loading of message. Due to BRAM contention, message expansion takes 99 clock cycles while the round operation takes 448 clock cycles.

Logic only

In the logic only version, the BRAM is replaced with three DRAMs and six registers (6.2.3). The message, round constants and hash values are stored in DRAMs and the working variables in registers. Using registers for working variables reduces the required clock cycles for round operation to 192 clock cycles. The number of clock cycles for message expansion increases to 196 clock cyles due to use of single port DRAM. Using a dual-port DRAM can reduce the clock cyles but would increase area significantly. Using approximately additional 100 slices, the throughput can be doubled but it would violate the area constraint.

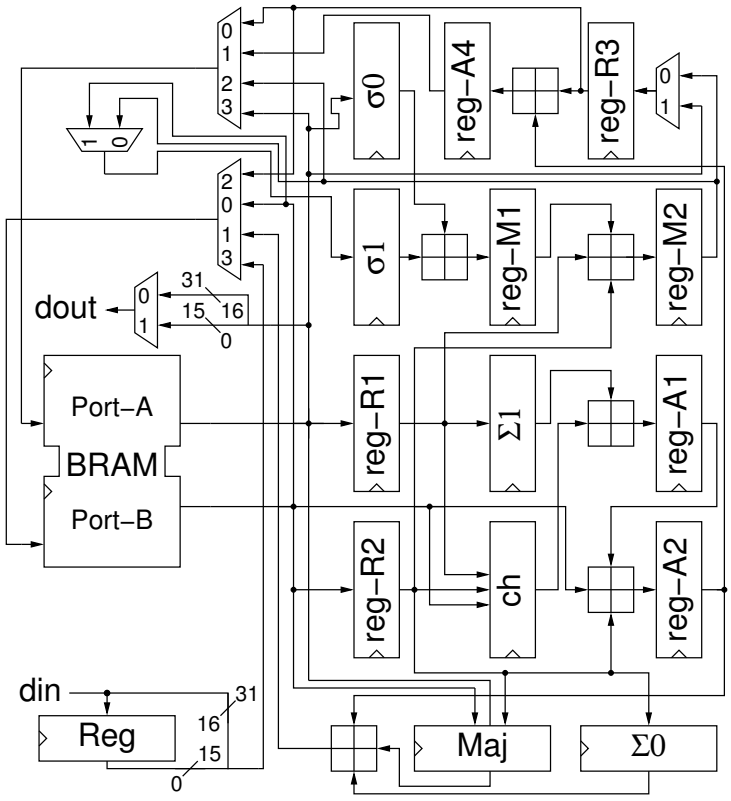

Figure 6.7: Datapath of SHA-256 using dedicated memory (BRAM)

6.2.4 Implementation Results

The implementation results of SHA-2 are compared with other SHA-3 finalist candidate implementations. Our SHA-256 achieves a throughput of 132.1 Mbps in case of logic only and 98.4 Mbps for the design using BRAM on Xilinx Spartan-3 device. Only BLAKE-256 [29] has a higher throughput which is about twice that of our SHA-256. This is due to higher latency of SHA-256 as compared to BLAKE-256 [29]. The same trend is observed across all the devices for both logic only and BRAM design.

6.3 Conclusions

We developed lightweight implementations of AES128 block cipher with 8, 16, and 32-bit datapath widths that can support feedback mode. Even with additional design constraints,

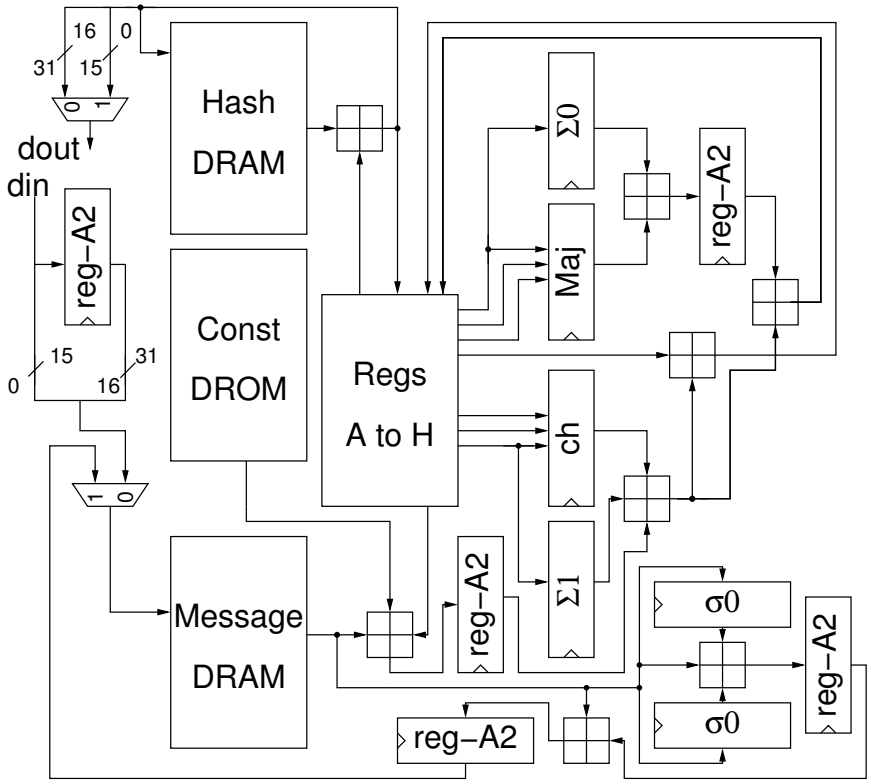

Figure 6.8: Datapath of SHA-256 using logic only

it can be seen that all three designs achieve higher performance both in terms of throughput and throughput/area ratio. Using these designs results, we conclude the following:

- Applying optimization techniques proposed in Chapter 5 can improve the design performance in terms of area, throughput and throughput over area.
- Datapth width below the natural width of a function would incur penalty in terms of area and latency
- Using embedded memory blocks (BRAMs) for SHA-256 degrades the performance.
- Only BLAKE-256 [29] outperforms SHA-256 across all device for both logic only and BRAM designs.

Table 6.4: Implementation results of SHA-256 compared with other implementations of SHA-3 candidates(TW= This Work)

Device	Version	Message Algorithm	Area (slices)	Block RAMs	Maximum Delay (ns) T	Long		Short	
						Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)
Xilinx Spartan-3 (xc3s50-5)	BRAM	BLAKE-256 [29]	549	1	8.05	219.3	0.40	205.9	0.375
		Grøstl [29]	594	1	7.65	122.4	0.21	61.9	0.104
		JH42 [29]	502	1	9.19	69.6	0.14	34.0	0.068
		Keccak [29]	627	1	8.90	32.5	0.05	32.3	0.052
		Skein [29]	498	1	10.65	19.7	0.04	10.0	0.020
		SHA-256[TW]	547	1	8.48	101.5	0.19	98.4	0.180
	Logic only	BLAKE-256 [29]	631	0	8.16	216.3	0.34	203.0	0.322
		Grøstl [29]	766	0	6.83	192.6	0.25	97.9	0.128
		JH42 [29]	558	0	10.05	63.7	0.11	31.2	0.056
		Keccak [29]	766	0	9.83	46.2	0.06	45.8	0.060
		Skein [29]	766	0	12.83	16.6	0.02	8.5	0.011
		SHA-256[TW]	745	0	8.52	137.8	0.19	132.1	0.177
Xilinx Spartan-6 (xc6slx4csg-3)	BRAM	BLAKE-256 [29]	152	1	5.63	313.8	2.06	294.5	1.938
		Grøstl [29]	271	1	4.80	195.0	0.72	98.7	0.364
		JH42 [29]	182	1	6.23	102.6	0.56	50.2	0.276
		Keccak [29]	127	1	5.07	57.0	0.45	56.8	0.447
		Skein [29]	182	1	7.19	29.2	0.16	14.8	0.081
		SHA-256[TW]	140	1	5.93	145.2	1.04	140.7	1.005
	Logic only	BLAKE-256 [29]	164	0	5.34	330.6	2.02	310.2	1.882
		Grøstl [29]	230	0	4.43	297.3	1.29	151.2	0.657
		JH42 [29]	156	0	6.14	104.2	0.67	51.0	0.327
		Keccak [29]	113	0	4.95	91.8	0.81	91.1	0.806
		Skein [29]	190	0	8.77	24.3	0.13	12.4	0.065
		SHA-256[TW]	227	0	5.74	204.6	0.90	196.0	0.864
Xilinx Virtex-6 (xc6vlx75T-1)	BRAM	BLAKE-256 [29]	163	1	5.06	348.7	2.14	327.3	2.008
		Grøstl [29]	241	1	4.09	229.1	0.95	115.9	0.481
		JH42 [29]	196	1	4.11	155.4	0.79	148.9	0.760
		Keccak [29]	129	1	3.84	75.2	0.58	74.9	0.580
		Skein [29]	207	1	6.00	35.0	0.17	17.8	0.086
		SHA-256[TW]	155	1	4.84	177.8	1.15	172.3	1.111
	Logic only	BLAKE-256 [29]	166	0	3.72	474.6	2.86	445.4	2.693
		Grøstl [29]	263	0	2.78	473.3	1.80	240.7	0.915
		JH42 [29]	171	0	3.96	161.5	0.94	154.9	0.906
		Keccak [29]	106	0	3.34	136.0	1.28	135.0	1.273
		Skein [29]	193	0	5.17	41.3	0.21	21.0	0.109
		SHA-256[TW]	238	0	3.86	304.2	1.28	291.5	1.225

Chapter 7: Evaluation of the CAESAR Hardware API for Lightweight Implementations

7.1 Introduction and Motivation

The Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR), aimed at developing a portfolio of new-generation authenticated ciphers surpassing the capabilities of current standards, such as AES-GCM [48], has moved to third round. The selection of authenticated ciphers for the new portfolio is based on three use-cases, namely lightweight applications, high-speed applications, and defense in depth. The use-case for lightweight applications includes performance of hardware implementations on resource constrained devices.

The Application Programming Interface (API) has a significant impact on the performance of any design. This is more true in case of hardware implementations. Hence, the CAESAR committee adopted a hardware API [49] which specifies the interface, communication protocol, and minimum compliance criteria. The CAESAR Hardware API is supported by a development package which includes VHDL code for universal pre- and post-processors for high-speed [50] and recently also for lightweight implementations. These processors are designed to simplify the complexity involved in making a cipher core design compliant with the API.

Designing lightweight implementations of cipher cores can be quite challenging and time consuming due to the difficulty in achieving a balance between area minimization of the datapath, the complexity of the controller, as well as performance. While the lightweight pre- and post-processors remove some of the additional burden of complying with the API, it is generally assumed that this comes at a cost of an increase in area consumption over merging their functionality with lightweight cipher cores. Integration of the protocol in an

existing state machine and re-using counters as well as storage should lead to a smaller area foot print.

In order to evaluate the penalty for using the generic lightweight pre- and post-processors of the CAESAR development package over integrated processors, we developed two lightweight implementations of KETJE-Sr, one with dedicated and one with integrated processors. These are the first lightweight implementations of KETJE-Sr. We also compare the overhead caused by the CAESAR API over the cipher core for both high-speed and lightweight implementations. For this, we developed the first lightweight implementation of an ASCON cipher core and attached it to the generic lightweight pre- and post-processors to make the design compliant with the CAESAR API and repeated this for a high-speed ASCON core with the high-speed processors.

7.2 Background

7.2.1 CAESAR Hardware API and Development Package

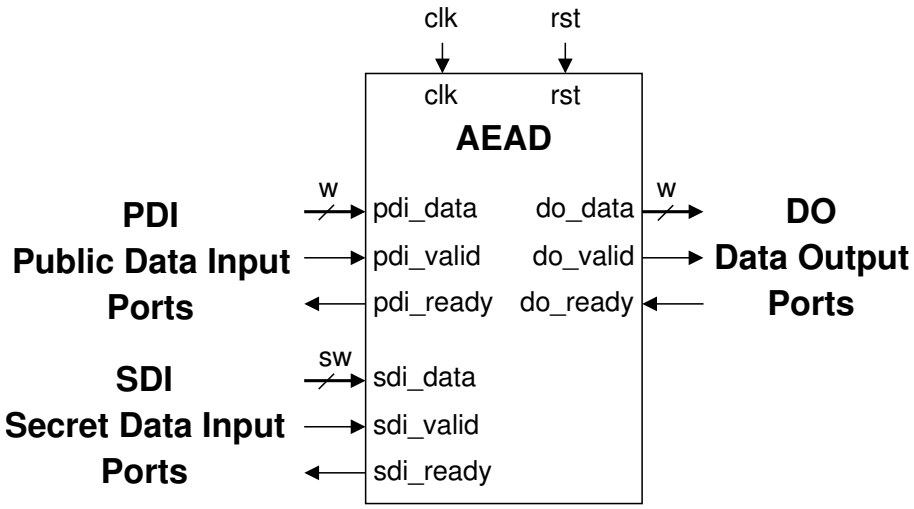

Figure 7.1: CAESAR API

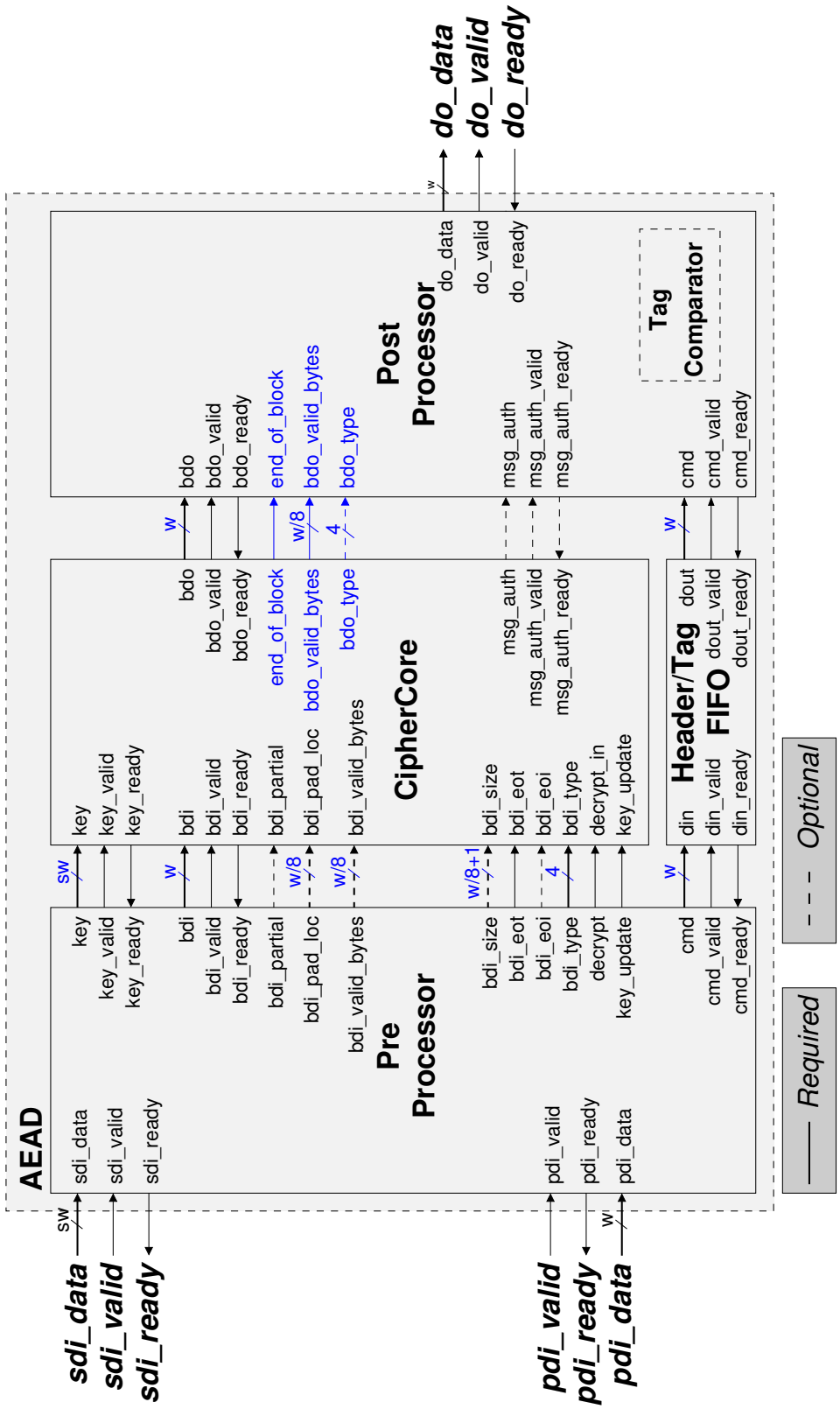

Figure 7.2: Lightweight CAESAR API block diagram

The CAESAR Hardware API (Fig 7.1) uses two input buses *Public Data Input (PDI)* and *Secret Data Input (SDI)* and one output bus *Data Output (DO)* along with their corresponding control signals *valid* and *ready*. For high-speed implementations, the permitted bus widths are $32 \leq w \leq 256$ for PDI and PDO and $32 \leq sw \leq 64$ for SDI. In case of lightweight implementations, they are limited to only three discrete values 8, 16, and 32. The CAESAR hardware API has two types of control words namely instructions and segment headers of widths 16 and 32 bits respectively. Instructions are used to specify the mode of operation, loading and activating key operations whereas segment headers contain information of the subsequent data i.e., data type, size of data etc.

Both lightweight and high-speed development packages have four modules namely *PreProcessor*, *CipherCore*, *FIFO*, and *PostProcessor*. A block diagram of the CAESAR lightweight hardware API along with internal modules and their interconnections is shown in Fig 7.2. The signals highlighted in blue are the differences between high-speed and lightweight modules. In this work, we consider only single-pass algorithms and therefore ignore all additional signals or modules for two-pass algorithm support.

The *PreProcessor* handles the CAESAR API protocol and provides data to the *CipherCore*. For high-speed implementations, it also pads data blocks when necessary. It is assumed that lightweight implementations perform the padding inside the *CipherCore*. In case the data is smaller than the width of the bus, it is zero padded to the bus width. The *PreProcessor* provides the necessary information to the *CipherCore* to perform padding, such as *bdi_valid_bytes* and *bdi_size* which indicate the location and number of valid bytes within the input data on the *bdi* bus. The *CipherCore* contains the cipher algorithm. Handling of the output protocol is performed by the *PostProcessor*. Additionally, the *PostProcessor* also performs tag comparisons for lightweight implementations. However, when an algorithm requires the tag for decryption, the tag comparison is performed within the *CipherCore* and the result is indicated through the two optional signals *msg_auth_valid* and *msg_auth_done* to the *PostProcessor*. The *FIFO* module provides a bypass path for headers from *Pre-* to *PostProcessor*.

7.2.2 Ketje

KETJE[51] CAESAR round 3 candidates whose primary use-case is "lightweight application". It is built on the KECCAK-p permutation, a round-reduced version of the KECCAK-f permutation [52] used in the new Secure Hash Standard (SHA-3) [53]. KETJE uses this underlying function in a mode called MONKEYWRAP which is based on MONKEYDUPLEX (Fig 7.3), a variant of the duplex construction [54]. There are four variants of KETJE: KETJE-JR, KETJE-SR, KETJE-MINOR, and KETJE-MAJOR which use KECCAK-p^*[200], KECCAK-p^*[400], KECCAK-p^*[800], and KECCAK-p^*[1600] respectively.

A round of the KECCAK-p permutation consists of five operations ($R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$). Algorithm 5 shows the pseudo-code for the round function in which A denotes the 5x5 state array and RC the round constant. B[x,y], C[x], D[x] are intermediate variables and the constant r[x,y] is the rotation offset. The width of A[x,y], B[x,y], C[x], D[x], and RC is b. Depending on the variant, the value of b is 8, 16, 32, or 64.

Algorithm 1 KECCAK-p Permutation Round Function

Input: (A,RC)
Output: A

1: θ step
   C[x] = A[x,0] $\oplus$ A[x,1] $\oplus$ A[x,2] $\oplus$ A[x,3] $\oplus$ A[x,4] $\forall$ x in 0...4
   D[x] = C[x-1] $\oplus$ rot(C[x+1],1), $\forall$ x in 0...4
   A[x,y] = A[x,y] $\oplus$ D[x], $\forall$ (x,y) in (0...4,0...4)
2: ρ step
   A[x,y] = rot(A[x,y], r[x,y]) $\forall$ (x,y) in (0...4,0...4)
3: π step
   B[y,2x+3y] = A[x,y] $\forall$ (x,y) in (0...4,0...4)
4: χ step
   A[x,y] = B[x,y] $\oplus$ ($\overline{\text{B[x+1,y]}}$ $\cdot$ B[x+2,y]), $\forall$ (x,y) in (0...4,0...4)
5: ι step
   A[0,0] = A[0,0] $\oplus$ RC

The θ step XORs each bit in the state with two bits from two different columns in the state array. This is achieved by using the temporary variables C[x] and D[x]. The bits in

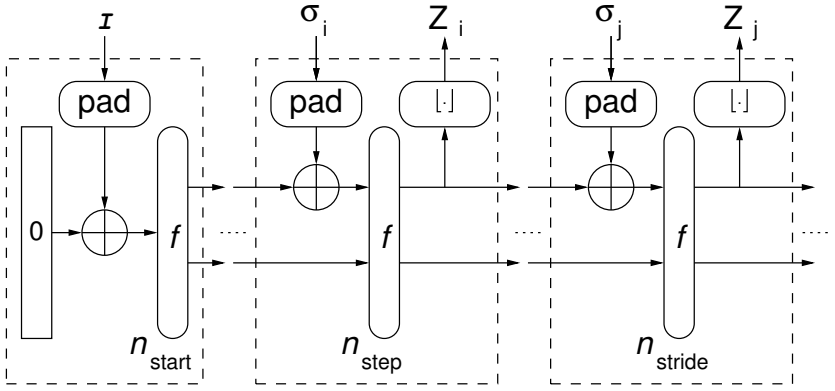

Figure 7.3: MonkeyDuplex construction

a word are rotated using one of 25 offsets in the ρ step. The value of the rotation offset for any word depends on its location in the state array. The words in the state array are rearranged in π. The χ step involves integer multiplication where each bit of the state is XORed with a non-linear function of two other bits. A round constant is added to word A[0,0] of the state array in the final ι step. As a round 3 tweak, a twisted permutation is added to Ketje*v*1 [55] version. The twisted permutation adds π and π^{-1} to Keccak-*p* which is just a reordering of bits in the state. The permutation function with the twisted permutation is referred to as Keccak-*p*$*$.

Start, *step*, and *stride* are the three stage in the MonkeyDuplex construction. In the first stage *start*, the state is initialized using key and Initialization Vector (IV), then the round function is iterated 12 times ($n_{start} = 12$). The processing of Associated Data (AD), plaintext, and ciphertext is done in the *step* stage. In this stage, the round function is iterated only once (n_{step}=1) for each block of AD, plaintext, or ciphertext. Finally, the tag is generated in the *stride* stage where the number of rounds is 6 ($n_{stride} = 6$).

7.2.3 Ascon

Ascon [56] is another CAESAR round 3 lightweight candidate which is also based on a permutation function like Ketje. It is built on a mode of operation called duplex sponge. In

order to enhance the security, ASCON employs stronger keyed initialization and finalization steps. There are two variants of ASCON referred to as ASCON-128, and ASCON-128a with the former being the primary recommendation. These two variants differ in terms of the rate at which data is processed and the number of rounds for processing the data.

The permutation function is made of three transformations called *constant-addition*, *substitution*, and *linear diffusion* (Algorithm 2). Each of the words x_i, t_i, y_i, z_i, and RC are 64-bit wide. In *constant-addition*, a round constant is added to the state. The *substitution layer* applies 64 parallel 5x5 S-boxes to the state. This layer can also be implemented using a bit-slice approach. Algorithm 2 shows the operations involved in the bit-slice implementation. For ease of understanding, we divide the *substitution layer* into four stages a, b, c, and d. The final step is *linear diffusion* which provides diffusion within each of the five words of the state. Diffusion is achieved by using right circular shifts and an XOR operations.

Algorithm 2 ASCON Round Function

Input: $(x_0, x_1, x_2, x_3, x_4$,RC)
Output: $(x_0, x_1, x_2, x_3, x_4)$

1: *Constant-addition*
$x_2 = x_2 \oplus \text{RC}$

2: *Substitution layer*
a) $t_0 = x_0 \oplus x_4$; $t_1 = x_1$; $t_2 = x_2 \oplus x_1$; $t_3 = x_3$; $t_4 = x_4 \oplus x_3$;
b) $y_0 = \overline{t_0} \cdot t_1$; $y_1 = \overline{t_1} \cdot t_2$; $y_2 = \overline{t_2} \cdot t_3$; $y_3 = \overline{t_3} \cdot t_4$;
$y_4 = \overline{t_4} \cdot t_0$;
c) $z_0 = t_0 \oplus y_1$; $z_1 = t_1 \oplus y_2$; $z_2 = t_2 \oplus y_3$; $z_3 = t_3 \oplus y_4$;
$z_4 = t_4 \oplus y_0$;
d) $x_0 = z_0 \oplus z_4$; $x_1 = z_1 \oplus z_0$; $x_2 = \overline{z_2}$; $x_3 = z_3 \oplus z_2$; $x_4 = z_4$;

3: *Linear diffusion layer*
$x_0 = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$;
$x_1 = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$;
$x_2 = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$;
$x_3 = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$;
$x_4 = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$;

Table 7.1: Comparison of KETJE and ASCON Parameters

Cipher	Variant	Key size (bits)	Nonce size (bits)	Tag size (bits)	Rate (bits)	State (bits)
KETJE	KETJE-JR	$\|K\|$	182-$\|K\|$	128	16	200
	KETJE-SR	$\|K\|$	382-$\|K\|$	128	32	400
	KETJE-MINOR	$\|K\|$	782-$\|K\|$	128	64	800
	KETJE-MAJOR	$\|K\|$	1582-$\|K\|$	128	128	1600
ASCON	ASCON-128	128	128	128	64	320
	ASCON-128a	128	128	128	128	320

$|K|$ →length of Key

ASCON's DUPLEX construction has four stages. In the first stage, the state is initialized with IV, key (K) and nonce (N). The state is updated by iterating the round function 12 times. Then the state gets XORed with $0^*||K$ which concludes the first stage. The second stage involves processing of AD where for each block of AD, the round function is repeated 6 times for ASCON-128 and 8 times for ASCON-128a. After processing the last block of AD, the state is XORed with $0^*||1$. Similarly, the plaintext or ciphertext gets processed in the next stage without the last XOR operation on the state. The last step is finalization which starts with an XOR of the state with $K||0^*$ followed by 12 round operations and finally an XOR with the key to generate the tag.

Various parameters for each variant of KETJE and ASCON are shown in Table 7.1.

7.3 Lightweight Designs

7.3.1 Design Decisions

The widths of the datapaths are chosen such that they equal the word size of their respective functions i.e., 16 for KETJE-SR and 64 for ASCON. The sizes of key, IV, and tag are fixed to 128-bits. We assume that key is changed rarely in case of lightweight applications. Therefore, the original key is preserved and only changed during key update operation. All our lightweight designs were developed with Xilinx Spartan-6 as the target device. This

allows us to leverage the architectural features of these FGPAs. Some of their 6-input Look Up Tables (LUTs) can be used as memory units referred to as Distributed RAMs (DRAMs) or as shift registers (SRLC32Es) [57]. We use DRAMs to store state, key, and constants. We take a hybrid approach for the design of the controllers i.e., a combination of traditional Finite State Machines (FSMs) using flip-flops and combinational logic and ROM-based FSMs using RAMs. Using RAMs for implementations of FSMs are found to be more efficient compared to the traditional approach [16], [13], [14]. Round operations typically repeat a short sequence of operations. Therefore, the control signals for round operations can easily be generated using a ROM-based FSM. No embedded resources such as block RAMs, DSP units etc, are used as they mask the true cost of implementing the designs. Additionally using these resources might cause a degradation in performance [45]. Furthermore, this allows porting of our designs to ASIC, however RAM cells would have to be used for the controller.

7.3.2 Lightweight Ketje-Sr

Two versions of Ketje-Sr were developed, one using the CAESAR API lightweight development package and one where we integrated the functionality of the pre- and post-processors in the controller of the cipher-core. Both designs use the same datapath architecture. The 16-bit datapath of Ketje-Sr is shown in Fig 7.4. A dual-port memory (RAM) with Port-A being read/write and Port-B ready-only is used to store the 400-bit state. The key is stored in two memory units with Most Significant Bytes (MSB) in RAMK1, and Least Significant Bytes (LSB) in RAMK2. Since key and IV are of fixed sizes, the $KeyPack$ function can be implemented using pre-stored values in RAMK1, and RAMK2, and by rearranging key bytes using register (reg-K). This approach allows us to implement the $KeyPack$ functionality with minimal cost in terms of area. Padding for message and AD are realized using three 8-bit wide 4-to-1 multiplexers.

The state memory is initialized using the output of RAMK1, RAMK2, IV and by applying the round function on the state for 12 rounds. The first step θ, involves computation

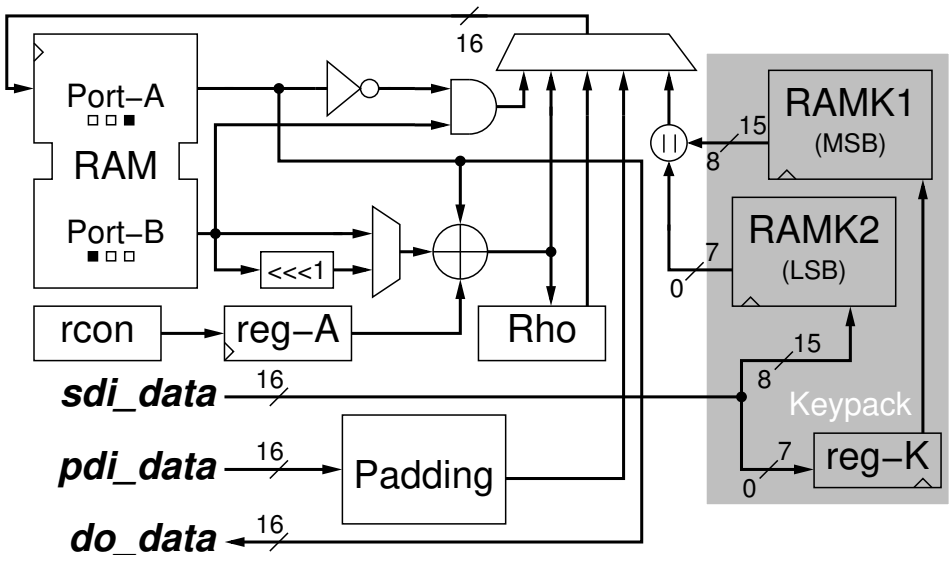

Figure 7.4: KETJE-SR datapath

of the temporary state variables $C_0, \ldots C_4$, and $D_0, \ldots D_4$. First, the five state variables of C are computed and stored in state memory. Using the stored five C variables, the five D variables are obtained and saved in state memory. In order to preserve space, some of the C variables' address locations are reused for storing D variables. The last operation in the θ step is the state matrix update using D. This operation is merged with the ρ and π steps. The rotations in ρ are performed using a variable rotator. In the χ step, each word of the state matrix is is a function of three other words. As the state memory has only two read-ports, it takes multiple clock cycles to update each word. Finally, a round constant is XORed with first word of the state. This operation is combined with the χ step by using register reg-A and the rcon module. Hence, no additional clock cycles are needed for this step. In total, each round operation requires 160 clock cycles. The additional operation introduced as the round 3 tweak is a twisted permutation which is not considered in our current designs. The datapath can support this additional operation without any changes as it can be accomplished through addressing. Therefore, only the controller needs to be modified. We believe that this does not alter the outcome of the current study. Hence no changes were made to the designs to include this tweak.

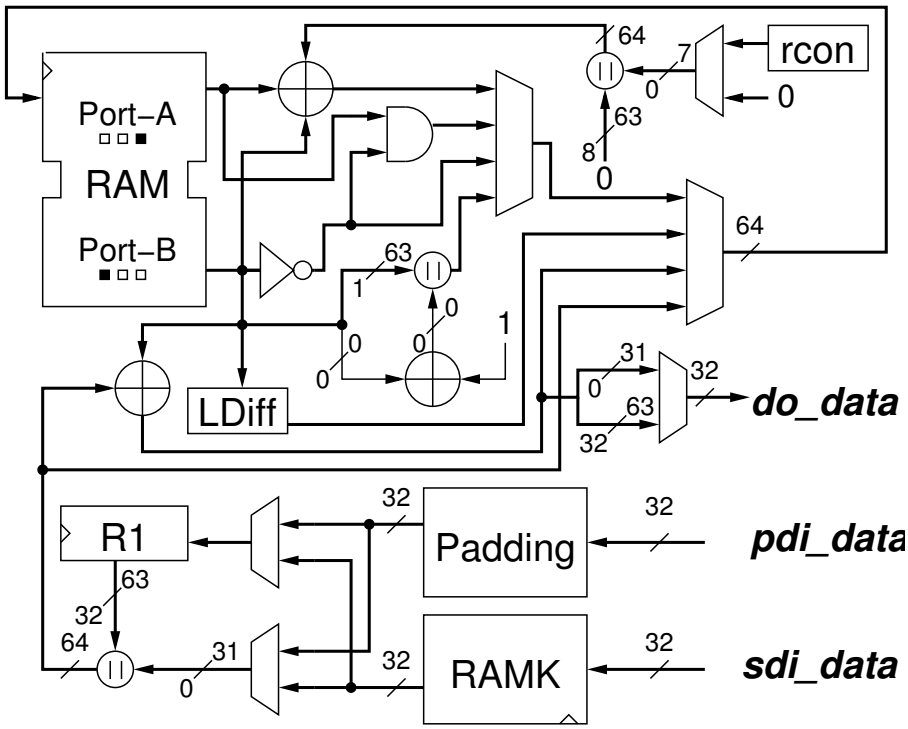

Figure 7.5: ASCON datapath

7.3.3 Lightweight Ascon

We developed architectures for both variants of ASCON using the CAESAR API lightweight development package. The widths of I/O buses are set at 32-bits. As the difference between ASCON-128, and ASCON-128a lies in the number of rounds and rate at which data is processed, they both have the same architecture for the datapath with a small exception in the round constant generation and initialization value. The controller required only a few minor changes.

The 64-bit datapath of ASCON is shown in Fig 7.5. The size of the nonce is fixed to 128 bits. A single-port memory (RAMK) is used to store the key along with state initialization constants. This removes the need for separate storage for these constants. Depending on the ASCON variant, the initialization constants in (RAMK) are adjusted accordingly. The padding for message and AD is accomplished using four 4-to-1 8-bit multiplexers. The 320-bit state is stored in a 64 x 10 dual-port memory with Port-A as read/write and Port-B

as read-only port. ASCON uses twelve 64-bit round constants. Except for the last byte, all other bytes are zero for each of these round constants. Assuming each of these last bytes are concatenation of two nibbles, all round constants can be generated by using two 4-bit registers, a 4-bit adder and a 4-bit subtractor. Different initial values for these 4-bit registers are used depending on the variant. The *constant-round* operation is combined with the *substitution layer* operation. Therefore no additional clock cycles are required for this operation.

The *substitution layer* is implemented using a bit-slice approach which is more efficient when the state is stored word-wise in a memory. Even though there are only eighteen operations involved, it takes 33 clock cycles due to contention on data ports of the state memory. The number of clock cycles can be reduced by adding additional storage units. The rotations in the *linear diffusion* step are implemented using two 5-to-1 multiplexers located inside the functional block *LDiff*.

The key is initially loaded into the memory (RAMK) and then into the state memory. The 32-bit register R1 acts as a buffer for the expansion of from the 32-bit input to the 64-bit datapath. Once the nonce N is loaded into the state memory, the round function is applied on the state 12 times for initialization. It takes 38 clock cycles for each round operation. The majority of the clock cycles in the round are due to the bit-slice implementation of the *substitution layer*. Apart from the clock cycles for round operations, five additional clock cycles are needed to perform the final XOR with the key to complete initialization. Similarly, five additional clock cycles are needed in the finalization as the state is XORed with $K||0^*$. The final XOR with K is performed during the generation of the tag. Hence no additional clock cycles are required.

7.4 Results

All the designs were implemented on a Xilinx Spartan-6 FPGA and optimized using the Automated Tool for Hardware EvaluatioN (ATHENa) [58]. All reported results were obtained after place-and-route. The primary optimization criteria is set to area with highest

Table 7.2: Area overhead High-Speed (HS) vs. LightWeight (LW) packages

Design	Top-level	Slices	LUTs	Filp-flops
LW Ascon	AEAD[1]	231	684	268
	CipherCore	196	606	212
Overhead		35	78	56
HS Ascon [59]	AEAD[2]	416	1282	792
	CipherCore	379	1033	529
Overhead		37	249	263

[1] ⇒ CAESAR LW Package; [2] ⇒ CAESAR HS Package

throughput to area ratio as the secondary optimization. We refer to block size as the rate at which plaintext, ciphertext, or AD are processed. For simplicity, we refer to the lightweight development package of the CAESAR hardware API as CAESAR LW package and for high-speed as CAESAR HS package. Table 7.2 shows the overheads for CAESAR LW package and CAESAR HS package. Table 7.3, contains results for all our implementations **(in bold)** as well as other lightweight implementation of CAESAR candidates available in the current literature [59].

The CAESAR LW package is worthwhile to consider if it satisfies two cases:

- Case 1: Smaller area foot print than CAESAR HS package.
- Case 2: Small overhead compared to an integrated controller for CAESAR API support.

Case 1: CAESAR LW vs HS packages: We evaluate this case by comparing our lightweight Ascon implementation with the high speed Ascon[1] implementation as both have the same I/O bus widths $w = 32$ but are using different CAESAR packages. To determine the area foot print, we implemented the designs with and without CAESAR API support. Using these results, we calculate the overhead for the corresponding packages as shown in Table 7.2. The overhead for using the CAESAR LW package on a lightweight design compared to the overhead of using the CAESAR HS package on a high speed design

[1]VHDL is available at `https://cryptography.gmu.edu/athena/index.php?id=CAESAR`

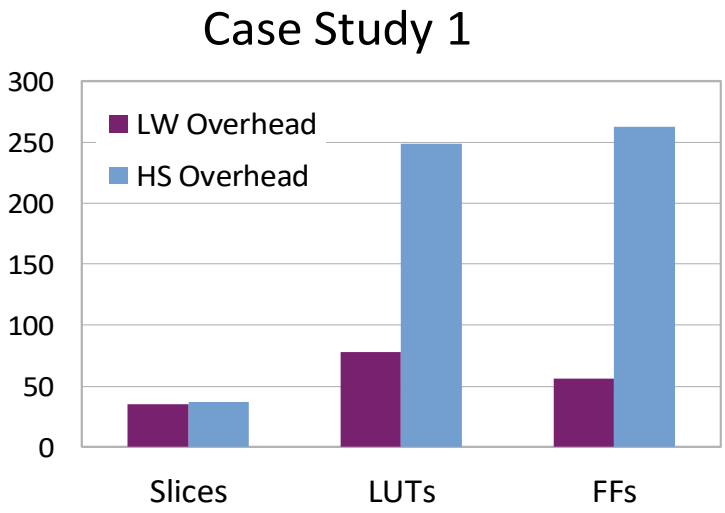

Figure 7.6: Comparison of CAESAR LW vs HS package overheads

is about 6% in terms of slices and about 69% in terms of LUTs and 78% for flip-flops (Figure 7.6). The required number of slices for high-speed is lower than expected based on the large number of LUTs and flip-flops. This may be due to higher slice utilization (i.e., more LUTs of the slices are used) and some merging of the logic for high-speed API with the cipher core. The higher count of flip-flops used in the high-speed design is due to the additional storage required for expansion of input data to the corresponding block sizes in the *preprocessor* which is not done in CAESAR LW. Considering the utilization of all three resources, we can clearly state that the CAESAR LW package has a smaller area foot print compared to the CAESAR HS package.

Case 2: Integrated vs CAESAR LW package APIs: In order to assess this case, we consider our two lightweight KETJE-SR implementations, one using an integrated controller and the other using the CAESAR LW package. In the first section of the Table 7.3, we calculate the overhead involved due to usage of the CAESAR LW package. As expected, there is a penalty for using the CAESAR LW package because some functionality for supporting the CAESAR API can be absorbed into the cipher controller. However, the overhead in terms of resource utilization (slices:15, LUTs:14, flip-flops:16) is very small which can be

Table 7.3: Implementation Results on Xilinx Spartan-6 FPGA

Design	State Size (bits)	Block Size (bits)	Key Size (bits)	Slices	LUTs	Flip-Flops	Frequency (MHz)	Throughput (Mbps)	Throughput/ Area (Mbps/slice)
KETJE-SR[1]	400	32	128	140	436	98	122.4	24.48	0.17
KETJE-SR[2]	400	32	128	155	450	114	120.1	24.03	0.16
Overhead				15	14	16			
ASCON-128[2]	320	64	128	231	684	268	216.0	60.10	0.26
ASCON-128a[2]	320	128	128	231	684	268	216.0	119.16	0.52
Joltik [59][3]	64	64	64	168	534	381	200.0	426.67	2.54
ACORN [59][4]	293	8	128	202	540	383	231.6	1,852.80	9.17

[1] ⇒ Dedicated CAESAR API; [2] ⇒ CAESAR LW Package; [3] ⇒ Not compliant to CAESAR API; [4] ⇒ Tweaked CAESAR HS Package

seen in Figure 7.7. For simplicity, separate counters are used for *sdi* and *pdi* buses in the *PreProcessor* of the CAESAR LW package. Therefore, using only a single counter for both buses could reduce the area overhead. When we consider the performance of these designs, there is only a 2% difference for throughput and a 6% difference for throughput by area ratio. These difference may vary with the cipher being considered.

Based on these two case studies,we believe it is worthwhile to use the CAESAR LW package in development of lightweight hardware designs to be compliant with CAESAR hardware API.

Since no other lightweight implementations of KETJE-SR and ASCON exist in the current literature, we compare our implementations between each other and with lightweight implementations of other CAESAR candidates. ASCON-128a clearly out performs both KETJE-SR designs by about 5 times for throughput and about 3 times in terms of throughput to area ratio. The KETJE-SR designs can be improved further by adding additional storage units as its design is much smaller than ASCON-128a.

Joltik [59] and ACORN [59] are two lightweight implementations of CAESAR candidates available in the literature. Joltik [59] has a much better throughput than our designs. We

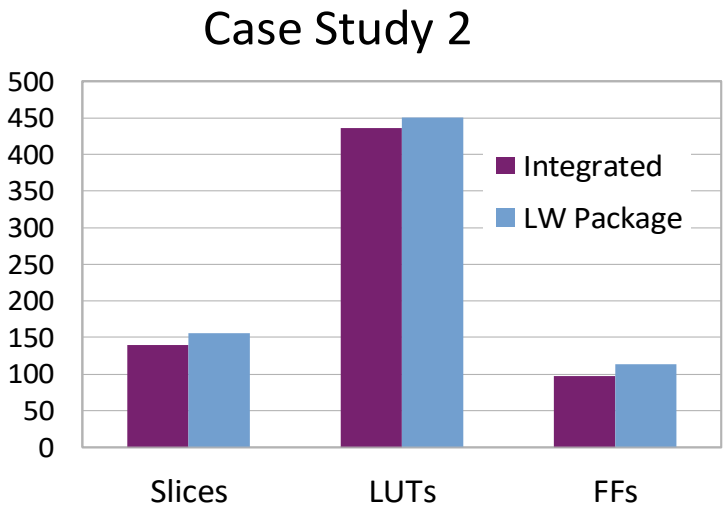

Figure 7.7: Comparison of integrated vs CAESAR LW package

believe this is due its smaller key size (64-bit) and its non-compliance with the CAESAR hardware API which adds additional cost. Furthermore, the design uses a custom I/O giving it an additional advantage. Another lightweight design is ACORN [59] which is based on a stream cipher. Lightweight implementations of authenticated ciphers which are based on stream ciphers achieve a very high throughput even comparable to that of high speed designs. But they have high latency for initialization which is not considered in the calculation of throughput and throughput/area ratio.

7.5 Conclusions

We justify the benefits of using the lightweight development package of the CAESAR hardware API with two case studies. As a part of these case studies, we developed the first lightweight FPGA implementations of Ketje-Sr, Ascon-128, Ascon-128a. Using Ascon as test case, we have shown that the CAESAR LW package has a very low area foot print and is about 69% smaller in terms of LUTs and 78% flip-flops compared to the CAESAR HS package. Furthermore, using Ketje-Sr with an integrated controller and Ketje-Sr with the CAESAR LW package, we demonstrate that the overhead of the CAESAR LW

package is negligible. We conclude that using the CAESAR LW package eases the burden of making a cipher compliant with the CAESAR API.

Chapter 8: Comparison of Multi-Purpose Cores of Keccak and AES

Cryptographic algorithms are used to provide authentication, integrity, confidentiality, and non-repudiation services required by security protocols such as IPSec [60], SSL [61] and TLS [62]. As no single cryptographic algorithm can provide all these services, a combination of algorithms is used. Furthermore, these protocols rely on pseudo random numbers for secret keys and nonces. Several of these services could be provided by a single secret key algorithm such as AES [44] through application of several modes of operation. This would be beneficial as fewer resources are required to implement one algorithm that can provide several cryptographic services than to implement many different algorithms.

While AES is a natural choice for an all-in-one implementation due to its popularity, Keccak [63], specifically Keccak's f-permutation, is also a very interesting option. Keccak is the winner of the competition for the next Secure Hash Algorithm (SHA-3). Its predecessors, SHA-1 and SHA-2 are being used widely. As a result, Keccak is very likely to be adopted by the general public. Its versatile f-permutation [64] allows it to operate in multiple modes to support various cryptographic services needs. Furthermore, the Keccak f-function is also the basis of two candidates of the cryptographic Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) namely Ketje and Keyak. Keccak has been shown to perform very well in hardware [65], [32], [66], [67] and [68].

As a result, we want to investigate cryptographic cores that can provide the following services: **Integrity** is provided through a **Hash Function** which takes a variable-length input message M and produces a fixed length output which is called hash H. The length of the message is denoted by $|M|$. **Authentication** and integrity can be provided by a **Message Authentication Code** (MAC). It takes the same inputs as a hash function and

additionally a secret key K. The output of the MAC is an authentication tag T.

Confidentiality, integrity, and authentication can simultaneously be provided by **Authenticated Encryption** (AE). AE schemes have the same inputs as MACs and generate, in addition to the authentication tag T, the encrypted message called cipher text C. Securely combining confidentiality and authentication using two separate algorithms has shown to be non trivial. Hence, it is advantageous to use a single algorithm that can provide both. Furthermore, several AE schemes also support to provide only authentication to other data that is associated with the message which requires integrity. Such a scheme is called Authenticated Encryption with Associated Data (AEAD) and its additional authenticated data input is AD. Some cryptographic algorithms require an initialization vector IV as an additional input. **PRNG** uses a random seed S as input and generates a random string R.

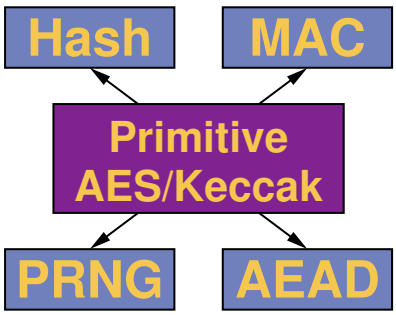

Figure 8.1: Various cryptographic services using same cryptographic primitive

8.1 Background

8.1.1 AES

Different modes of operation have to be used for AES to function as a Hash, MAC, AEAD, and PRNG. We chose AES-Hash [69] as our hashing mode which has been proposed to NIST as a mode of operation. It is a variant of Davies-Meyer [70] and uses Rijndael

with a block size of 256-bit and a 256-bit key. The Cipher based Message Authentication Code (CMAC) [71] is a NIST recommended mode of operation for authentication and is equivalent to OMAC1, a variation of One-Key CBC-MAC (OMAC). For AE schemes, NIST recommends Galois/Counter Mode (GCM) [48]. Fortuna [72] was developed as a cryptographically secure PRNG mode. Table 8.1 shows the parameters for each mode of AES.

Table 8.1: AES / Rijndael* Modes (Rd. = Number of rounds)

Operation	Mode	Block	Key	Rd.	Inputs	Outputs
Hash*	AES-Hash	256	N/A	14	$\|M\|$, M	H
MAC	CMAC	128	128	10	$\|M\|$, M, K, IV	T
AEAD	GCM	128	128	10	$\|M\|$, M, K, IV, $\|AD\|$,AD	T, C
PRNG	Fortuna	128	N/A	14	S	R

Table 8.2: Keccak Modes (Rd. = Number of rounds)

Operation	Mode	Block	Key	Rd.	ρ	Inputs	Outputs
Hash	Sponge	1600	N/A	24	1088	$\|M\|$, M	H
MAC	Sponge	1600	128	24	1088	$\|M\|$, M, K, IV	T
AEAD	Duplex	1600	128	12	1344	$\|M\|$, M, K, IV, $\|AD\|$,AD	T, C
PRNG	Duplex	1600	N/A	12	1344	S	R

8.1.2 Keccak

Keccak [64] is a family of cryptographic hash function which maps a variable-length input to variable-length output using a fixed length permutation called f-permutation. The f-permutation operates on $b = r + c$ bits, where b, r, and c, are called width, bit-rate

and capacity respectively. All the cryptographic services we presented in this paper for Keccak are based on two underlying constructions. The first construction is called Sponge construction where a variable length input is mapped to a fixed length output. The second mode is called Duplex construction which allows alternation of inputs and outputs at the same bit rate. Table 8.2 shows the parameters for each mode of Keccak we explore in this paper.

The functions Hash and MAC are built using the sponge construction. The message M is padded such that it is a multiple of the block size (r). In MAC mode, Key and IV are processed as if they are normal blocks of a message. With the state after processing Key and IV as the initial state, the message is hashed to produce MAC as the final result.

The modes PRNG and AEAD are built using the Duplex construction. In PRNG mode, a nonce is used as an initial seed. Pseduo random bits are generated after the processing of the initial seed. The maximum number of pseudo random bits that can be produced is limited by the bit rate (r). When more random bits are needed, more calls to the f-function must be made with an empty block as input. The AE mode is based on the DuplexWrap construction which, in turn is built on top of the Duplex construction for encryption. One could also call this mode of operation stream cipher mode as it generates the key stream for encrypting the data. The key stream (Z_0) is produced after processing Key and IV. The key stream is XORed with a message to produce the cipher text. Also, the message serves as the input to the state for generating the subsequent key stream bits Z_1, Z_2,...Z_{n-1}. This process is repeated until the last block of the message. The output after processing the last block is called Tag.

8.1.3 Padding

Most of the modern cryptographic hash functions split a message into blocks of fixed size for processing. Usually messages are not a multiple of the block size. Hence additional bits are appended. This process is called padding. Keccak's Sponge and Duplex construction pad data differently. In Sponge construction, only the last block of message is padded. In

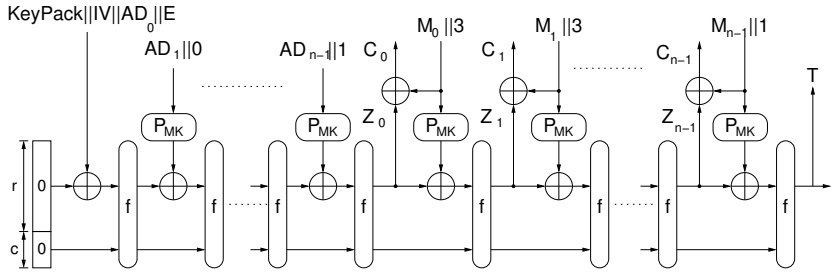

Figure 8.2: Authenticated Encryption Mode in Keccak

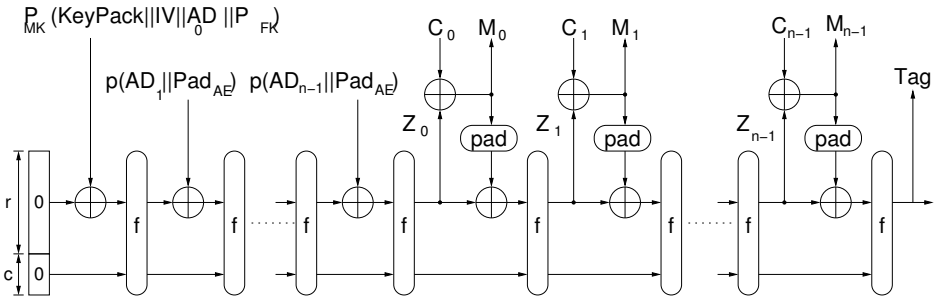

Figure 8.3: Authenticated Decryption mode in Keccak

Duplex construction, additionally to the Sponge padding, each block $< r$ and is padded to reach r. As a result, while the same rate (r) is used for both constructions, the data being encrypted/hashed by them is consumed at a slightly different speed. As a result, to simplify the design, we assume that all input data must be a multiple of a byte. For more details about padding, we refer to [63], and [73].

8.2 Design Decisions

We designed low-area (LA) architecture for each Keccak and AES. These two architectures support all modes for Hash, MAC, AEAD, and PRNG. Analyzing the results of these implementations led us to additionally designing two dedicated implementations of each Keyak and AES-GCM. The datapath width of the LA architecture for AES is 32 bits as this is the width of the largest single operation: MixColumn. For Keccak we used a datapath width of 64 bits which is the width of a word in Keccak. All architectures have

the secondary design goal of high throughput to area ratio. We assume that all input data must be a multiple of a byte. Padding of messages is performed in hardware. Input data is assumed to be zero padded if not a multiple of the I/O width. For more details about padding, we refer to [63], and [73]. Keccak-r is chosen to be 1088-bit, which is the recommended bit rate that would provide the same level of security as SHA-256. To keep the design simple, the size of key, IV and seed are fixed to 128 bits.We consider that 128-bit key and IV provide adequate security margin. The interface used is based on [46] and [47]. The data input (DI) and outputs (D0) are designed to operate with FIFOs. The width of these interfaces w is set to 16-bit

8.3 Low Area Architecture of AES

The LA AES datapath has two dual-port 32-bit wide RAMs with dedicated read and write ports which are used to store various inputs and state variables. These RAMs are actually a combination of four 8-bit wide RAMs which allows splitting of 32-bit words into four individual bytes. This way of storing the state allows to perform the shift-row operation by addressing. It takes four clock cycles to perform one round of AES. In case of AES-Hash, it takes eight clock cycles for one-round due to the 256-bit block size. The original key (K), round key (K_{rnd}), and two subkeys used in AES-CMAC are stored in an additional RAM. The multiplication in AES-GCM is performed using a 128x2 multiplier and two 128-bit registers. It takes 64 clock cycles to perform one 128x128 multiplication. The dedicated AES-GCM design is a reduced version of multi-purpose core without additional hardware to support other modes.

8.4 Low Area Architecture of Keccak

Two dual-port distributed RAMs with dedicated read and write ports are used to store the state matrix along with all the other state variables. The state variables C and D in the θ step of f-function are computed using a register and a couple of multiplexers. It takes

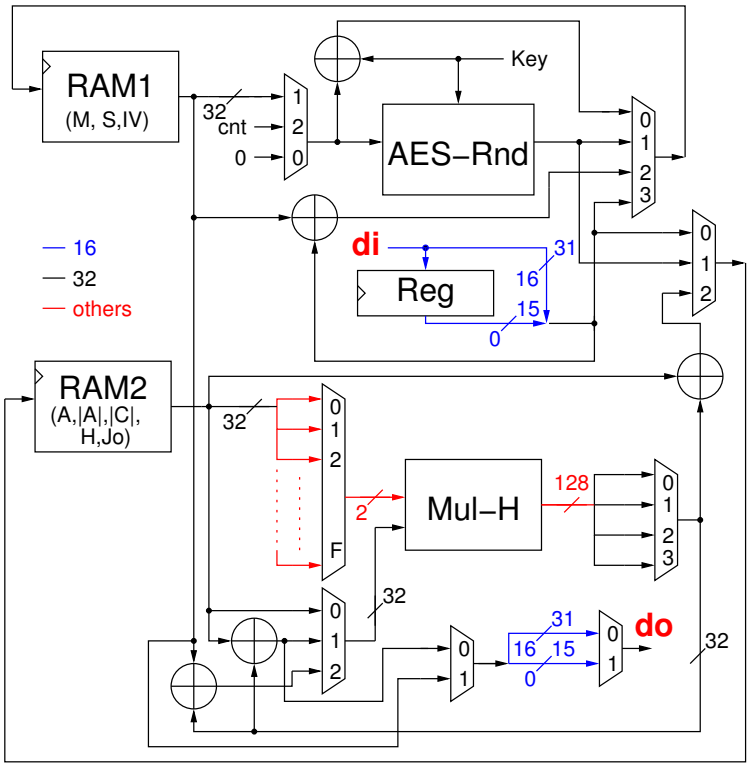

Figure 8.4: Low area datapath of AES

14 clock cycles to compute the 5 state variables C_0 to C_4 and 6 clock cycles to compute other 5 state variables D_0 to D_4. These 10 state variables are stored in both RAMs as they are required for both even and odd state words. The π step is performed by means of addressing the words. The 25 different cyclic rotations in ρ step are performed using three pipelined 4x1 multiplexers and the χ step using three registers. All together, these three steps are computed in 39 clock cycles. To conserve resources only 6-bits of each of the 24 round-constants in ι step are stored in memory as the remaining bits are all zeros. No additional clock cycles are required for ι step. In total it takes 58 clock cycles to perform one round operation. Through scheduling the operations of two consecutive rounds, the total clock cycles for 24 round is reduced to 1323 from the expected 1392 clock cycles. An additional single-port RAM is used to store the Key, Seed and IV. Since the sizes of Key, Seed and IV are fixed, padding for them requires minimal additional logic. Hence it is

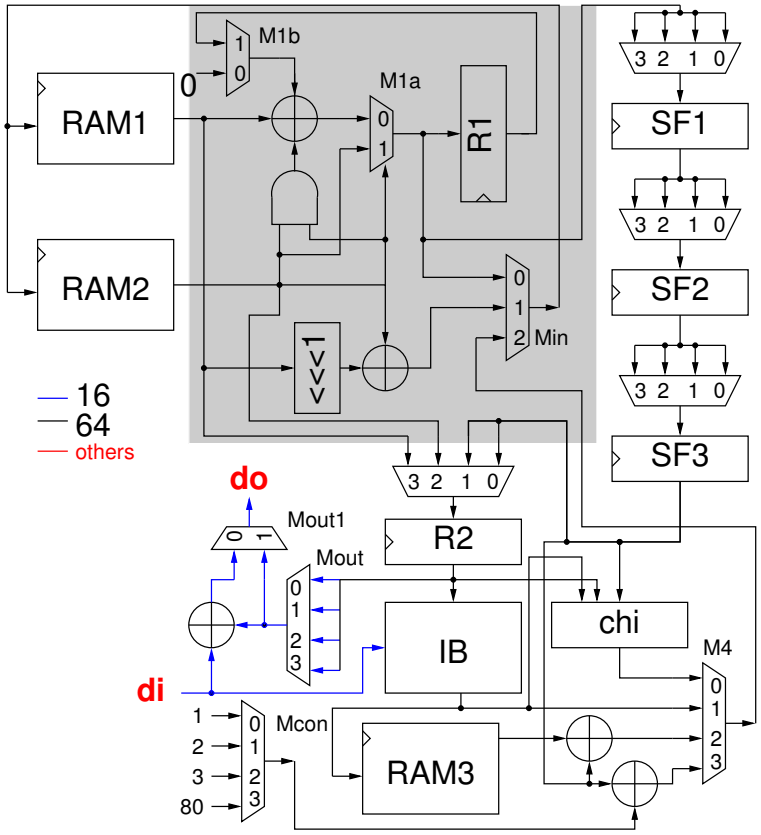

Figure 8.5: Low area datapath of Keccak

included in the core. The dedicated Keyak design is derived from the multi-purpose Keccak with minimal change.

8.5 Results

All of our results are after place-and-route and were generated using Automated Tool for Hardware EvaluatioN (ATHENa) [58] with Xilinx ISE 14.7 and Quartus II 13.1. Throughput (TP) is calculated for long messages i.e the number of clock cycles required for initialization, preprocessing of Key and IV are assumed to be zero. None of our designs utilize embedded resources for ease of comparison. Using embedded resources improves the performance of AES, however it would degrade the performance of Keccak as shown in a previous study [45]. Some results reported by others, especially the ones from the

industry, may include them as not all information is provided. We implemented our designs on Xilinx Virtex-5, Spartan-6, Virtex-6, Artix-7, and Virtex-7 and Altera Cyclone-IV and Stratix-IV FPGAs. Detailed results for high-speed implementations on Xilinx Virtex-7 and for low-area implementations on Xilinx Artix-7 are shown in Table 8.3. All performance comparisons are made with respect to TP/Area.

Figure 8.6 shows the performance of our multi-Keccak implementations relative to the multi-AES implementations for all modes of operation on all devices. In almost every case, the performance of multi-Keccak is much better, up to 14 times, than of multi-AES. In terms of throughput, this is not surprising as the width of the Keccak datapath in high-speed designs is 12.5 times wider than AES and 2 times for low-area designs while the number of rounds is similar. However, this increase in datapath width does not come with an increase in area, leading to much better TP/Area results. This is due to the fact that AES modes of operation have vastly different underlying characteristics. As a result, resource sharing is not possible. On the other hand, the primary difference between Keccak modes is how the input blocks are formatted. Hence, Keccak requires minimal additional resources. In case of low-area designs the performance of Keccak in AEAD mode stands out. The reason is the number of clock cycles required for the AES-GCM multiplier.

Our dedicated implementations of Keyak outperform our AES-GCM implementations in a similar way as multi-Keccak outperforms multi-AES (Fig. 8.7). For low-area implementations the relative performance of Keyak is higher than multi-Keccak in AEAD mode. However, for high-speed designs the opposite is true. That is due to the fact that multi-AES employs a dual-core AES while AES-GCM has only a single AES core.

Comparison with results from literature, which are on Xilinx Virtex-5, are reported in Table 8.4. As there are no reported results of a design which can be operated in all the four modes of operations, we compare our results with designs that utilize block ciphers operating in a specific mode. When comparing our low-area designs in Hash mode, our Multi-Keccak performs better than [29] but not against [30]. We believe that the reduction of performance is due to support for other modes. In case of dedicated modes, our Keyak

Table 8.3: Results of AES and Keccak Implementations

Mode	Design	Area (Slices)	Freq. (MHz)	TP (Gbps)	TP/Area (Mbps/ Slices)
Xilinx Artix-7 FPGA([74])					
Hash	Multi-AES	2852	107.53	1.835	0.643
	Multi-Keccak	2299	115.87	5.253	2.285
MAC	Multi-AES	2852	107.53	1.251	0.439
	Multi-Keccak	2299	115.87	5.253	2.285
AEAD	Multi-AES	2852	107.53	2.502	0.877
	Multi-Keccak	2299	115.87	12.978	5.645
PRNG	AES-PRNG	2852	107.53	1.835	0.643
	Multi-Keccak	2299	115.87	12.978	5.645
Dedicated AEAD	AES-GCM	1425	172.56	2.008	1.409
	Keyak	2173	133.39	14.940	6.875
High-Speed Designs on Xilinx Virtex-7 FPGA ([74])					
Hash	Multi-AES	3061	188.18	3.212	1.049
	Multi-Keccak	2495	206.70	9.370	3.756
MAC	Multi-AES	3061	188.18	2.190	0.715
	Multi-Keccak	2495	206.70	9.370	3.756
AEAD	Multi-AES	3061	188.18	4.380	1.431
	Multi-Keccak	2495	206.70	**23.150**	9.279
PRNG	AES-PRNG	3061	188.18	3.212	1.049
	Multi-Keccak	2495	206.70	23.150	9.279
Dedicated AEAD	AES-GCM	1455	352.98	4.107	2.823
	Keyak	2444	258.40	**28.941**	11.841
Low-Area Designs on Xilinx Artix-7 FPGA[TW]					
Hash	Multi-AES	629	82.83	0.166	0.263
	Multi-Keccak	264	152.23	0.125	0.474
MAC	Multi-AES	629	82.83	0.189	0.301
	Multi-Keccak	264	152.23	0.119	0.451
AEAD	Multi-AES	629	82.83	0.074	0.117
	Multi-Keccak	264	152.23	0.274	1.037
PRNG	Multi-AES	629	82.83	0.379	0.602
	Multi-Keccak	264	152.23	0.280	1.060
Dedicated AEAD	AES-GCM	548	71.09	0.630	0.115
	Keyak	260	177.87	0.136	1.231
Xilinx Virtex-7 FPGA					
Hash	Multi-AES	532	169.38	0.339	0.637
	Multi-Keccak	267	306.84	0.252	0.945
MAC	Multi-AES	532	169.38	0.387	0.728
	Multi-Keccak	267	306.84	0.240	0.899
AEAD	Multi-AES	532	169.38	0.151	0.283
	Multi-Keccak	267	306.84	0.438	1.431
PRNG	Multi-AES	532	169.38	0.774	1.455
	Multi-Keccak	267	306.84	0.564	2.113
Dedicated AEAD	AES-GCM	521	153.53	0.136	0.262
	Keyak	272	414.08	0.745	2.739

Table 8.4: Comparison of our designs with other implementations on Xilinx Virtex-5 (TW = This Work)

	Mode	Design	Area (Slices)	Freq. (MHz)	TP (Gbps)	TP/Area (Mbps/ Slices)
High-Speed	Hash	Multi-AES [74]	2871	203.29	3.470	1.208
		Multi-Keccak [74]	2805	163.92	7.431	2.649
		Keccak[32]	1395	281.84	12.777	9.16
	MAC	Multi-AES [74]	2871	203.29	2.366	0.824
		Multi-Keccak [74]	2805	163.92	7.431	2.649
		GMAC[75]	9405	120.17	15.382	1.636
	Dedicated AEAD	AES-GCM [74]	1089	283.53	3.299	3.030
		AES-GCM[76]	678	335.00	2.250	3.319
		AES-CCM[77]	490	274.00	1.525	3.112
		Grøestl/AES[78]	3102	233.00	3.848	1.240
		Keyak [74]	2357	243.96	27.324	11.593
Low-Area	Hash	Multi-AES [TW]	478	131.23	0.262	0.549
		Multi-Keccak [TW]	318	257.00	0.211	0.665
		Keccak[29]	275	251.25	0.118	0.430
		Keccak[30]	393	159.0	0.864	2.198
	Dedicated AEAD	AES-GCM [TW]	351	130.87	0.116	0.331
		AES-GCM[76]	247	393.00	0.230	0.931
		AES-CCM[77]	214	272.00	0.363	1.696
		Keyak [TW]	259	281.29	0.506	1.954

performs better that [76], [77].Unfortunately, our dedicated AES-GCM does not perform as well as [76] and [77]. However, their implementation details are not known.

8.6 Conclusion

Overall, our Multi-Keccak design has a much better TP/Area than our Multi-AES design by about a factor of 4 across all functions and FPGAs as can be seen in Fig. 8.6. Also, the throughput of Keccak exceeds AES's on most FPGAs. The maximum throughput for Multi-Keccak AEAD is **23.2 Gbps** on Virtex-7 and **28.7 Gbps** on Stratix-IV. Multi-AES in GCM mode achieves **4.4 Gbps** and **5.6 Gbps** on the same devices respectively. In case of dedicated cores, the maximum throughput for Keyak and AES-GCM are **28.9 Gbps** and **4.1 Gbps** on Virtex-7 respectively. All in all, this clearly shows that Keccak is more suitable than AES as a basis for multi-service functions.

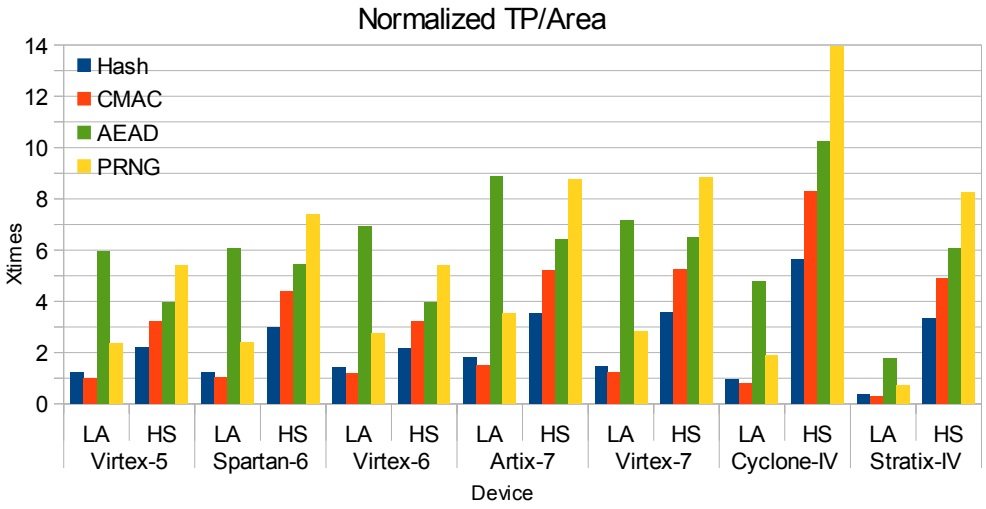

Figure 8.6: Performance improvement of multi-Keccak over multi-AES for specific modes of operation

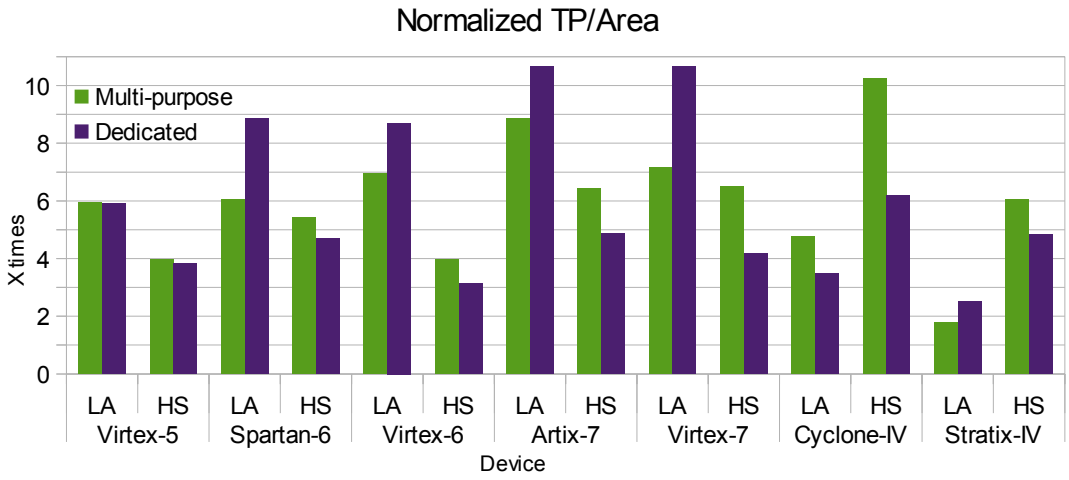

Figure 8.7: Performance improvement of dedicated and multi- purpose Keccak over corresponding AES cores for AEAD

Chapter 9: Lightweight AES IP Core for ASCIs

9.1 AES-LightWeight IP Core Features

The AES-LightWeight IP Core is designed for ASIC technology with small area footprint. We developed this core using an 8-bit architecture of AES block cipher.

This AES core is developed such that it can perform both encryption and decryption for all modes specified in Table 9.2. In addition to this, the core also supports two key lengths 128 and 256 bits. To keep the area foot low, the SubBytes and InvSubBytes are implemented using combinational logic with some resource sharing. Furthermore, SubBytes are shared between both for key expansion and encryption/decryption modules. A high level overview of the IP core is shown in Figure 9.1.

The AES LightWeight IP Core consists of two independent modules: Encryption/Decryption (EncDec) and Key Expansion (KeyExp). These modules are clearly separated in order to allow an implementer to trade area for security. For instance, in an area constrained design, one may want to apply the side-channel countermeasures only to the EncDec module.

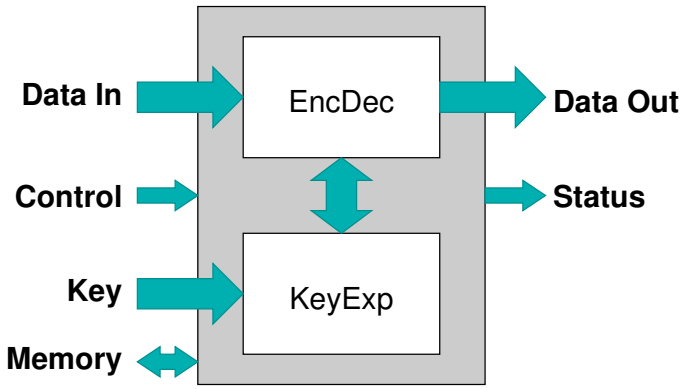

Figure 9.1: Overview of AES LightWeight IP Core

9.1.1 Interface and Modes of Operation

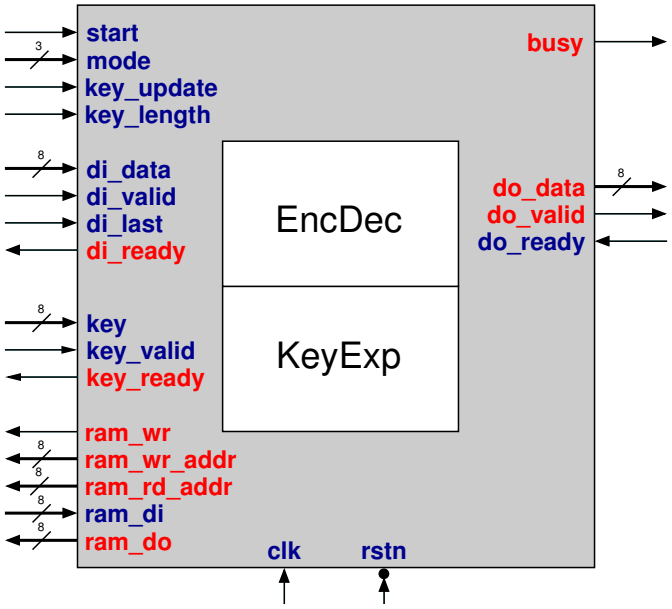

Figure 9.2: The AES LightWeight IP core interface diagram

The AES LightWeight IP core interface diagram is shown in Figure 9.2. External memories connected to the core are assumed to have a **registered output**, i.e., output data is available one clock cycle after a read (*di_ready*) was issued. Signals used to communicate between external control modules and the AES LightWeight core are shown in Table 9.1. Input and output ports are shown in **blue** and **red**, respectively. Operational codes for the *mode* signal are shown in Table 9.2. With the exception of ECB, all other modes require an IV block. This means that the first block prior to data has to be an IV block.

A standard operation of the core begins with a key expansion operation initiated by asserting *key_update* and setting the length of the key using *key_length*. The generated round keys are stored in an external memory, which can be read by the EncDec module. While the round keys are being generated, the status signal *busy* is set high to prevents the AES core from processing any data until all round keys are updated. The status signal

Table 9.1: Interface Signals

Group	Signal	I/O	Description
Global	clk	in	Clock signal
	rstn	in	Asynchronous reset active low
Control	start	in	Start encryption/decryption
	mode	in	Modes of operation (see Table 9.2)
	key_update	in	Update key
	key_length	in	Key length (1=256-bit, 0=128-bit)
Status	busy	out	Core busy
Data In	di_data	in	Input data bus
	di_valid	in	Input is available
	di_last	in	Indicates last byte of input
	di_ready	out	Ready to accept input
Data out	do_data	out	Output data bus
	do_valid	out	Output is available
	do_ready	in	Ready to accept output
Key	key	in	Key data bus
	key_valid	in	Key is available
	key_ready	out	Ready to accept key
Memory	ram_di	in	Input data from RAM
	ram_do	out	Output data to RAM
	ram_wr	out	RAM write enable
	ram_rd_addr	out	RAM read address
	ram_wr_addr	out	RAM write address

Table 9.2: Modes of Operation

Opcode/Mode	Operation	IV
000	ECB-Encryption	No
001	ECB-Decryption	
010	CBC-Encryption	Yes
011	CBC-Decryption	
100	CFB-Encryption	Yes
101	CFB-decryption	
110	CTR	Yes
111	OFB	Yes

is also set high while the EncDec is processing data, which prevents a key update. This safeguards keys from being overwritten by new keys while they are in use.

EncDec begins processing with a *start* signal, and setting the mode of operation (Table 9.2) through *mode*. Depending on the mode of operation, the data is loaded in proper

order. *For example:* In CTR-Encryption mode, IV is loaded initially, followed by plaintext blocks in order. The end of data (plaintext/ciphertext) is specified by asserting *di_last* high for the last byte of last block.

9.2 Datapath

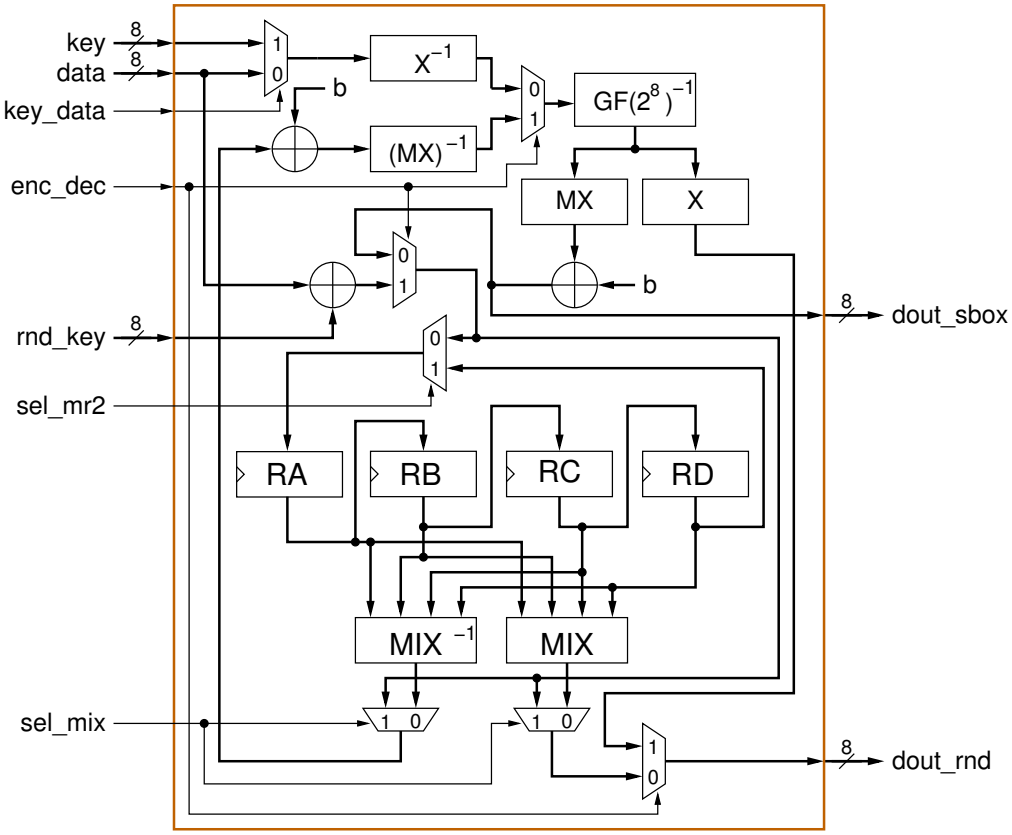

Figure 9.3: 8-bit datapath of AES round

The Figure 9.3 shows the 8-bit datapath of the round operation. The signal `key_data` allows the sharing of SubBytes resources between key expansion and encryption/decryption operations. The signal `enc_dec` enables switching between encryption and decrption operations. Since the datapath width is 8-bits which is less than the natural width of AES

(32-bit due to Mixcolumn/InvMixColumn), four 8-bit registers (RA, RB, RC, and RD) are needed for MixColumn/InvMixcolumn operations.

The state is stored using sixteen 8-bit registers (R1 to R15) as shown in Figure 9.4 These registers are grouped into four and within each group they are connected in shift register configuration where the data from one register is feed into next one. This arrangement is considered due to the shift row operation of AES and to reduce the size of output multiplexer. Similarly for IV, we use the same construction but with an 8-bit adder for counter mode as shown in Figure 9.5.

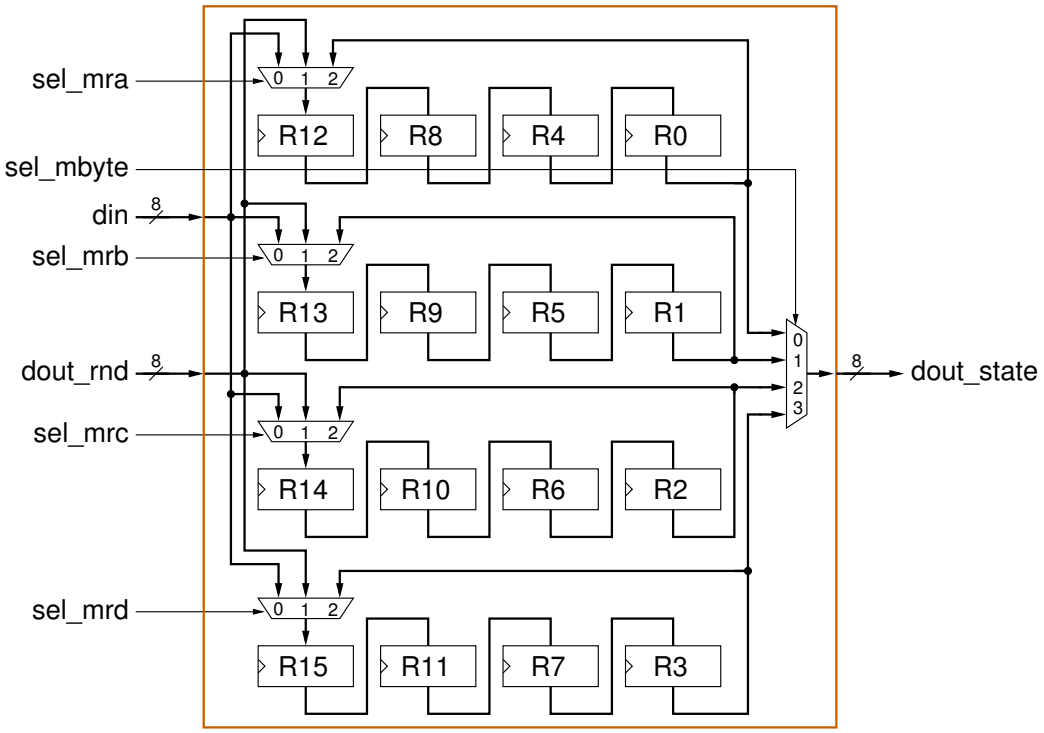

Figure 9.4: AES state using sixteen 8-bit registers

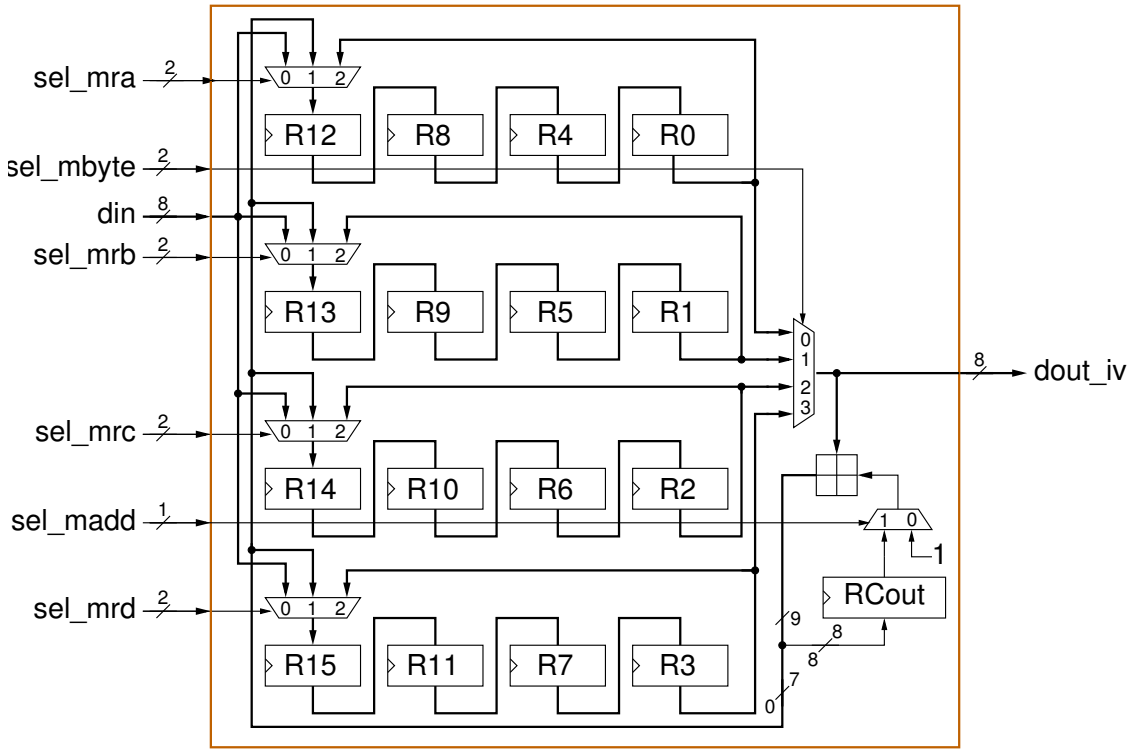

Figure 9.5: Storing IV using sixteen 8-bit registers

9.3 Design Performance

9.3.1 Latency

The latencies of AES encryption and decryption operations are based on the minimum number of clock cycles required to compute a single AES data block. The Table 9.3 lists the formulae for calculating the latencies of different operations supported by our design for processing N blocks. The latencies of Key Expansion are based on the minimum number of clock cycles required between the rising edge of *key_update* and the falling edge of the *busy* output.

Table 9.3: Operational Latency

Key length		**128-bit**		**256-bit**	
Key Expansion		321		417	
AES		**Encryption**	**Decryption**	**Encryption**	**Decryption**
MODE	**ECB**	339·N	348·N	467·N	480·N
	CBC	$17 + 339{\cdot}N$	$17 + 348{\cdot}N$	$17 + 467{\cdot}N$	$17 + 480{\cdot}N$
	CFB	$17 + 323{\cdot}N$	$17 + 323{\cdot}N$	$17 + 451{\cdot}N$	$17 + 451{\cdot}N$
	CTR	$17 + 323{\cdot}N$	$17 + 323{\cdot}N$	$17 + 451{\cdot}N$	$17 + 451{\cdot}N$
	OFB	$17 + 323{\cdot}N$	$17 + 323{\cdot}N$	$17 + 451{\cdot}N$	$17 + 451{\cdot}N$

9.4 Implementation Results

Our design is synthesized using Synopsys Design Vision and SAED_90nm library. We measure the area in terms of Gate Equivalents (GE) and throughput at maximum frequency (f_{Max}). The results for our ASIC implementation are summarized in the Table 9.4 for each mode and key sizes. The area foot print of the design is 6,952 GEs with a maximum frequency of 309.60 MHz. The highest throughput of 12.269 Mbps is observed for CTR/CFB/OFB modes with 128-bit key for encryption and lowest in case of ECB/CBC mode with 256-bit key for decryption. This is expected as decryption take more clock cycles than encryption and with 256-key you have more rounds.

For comparisons, we only considered cores which supports all five modes same as our IP Core. [79] is one such core implemented on 65nm technology. This core consumes about 6000 GEs and achieves throughput of 0.58 Mbps and 0.43 Mbps with 1 MHz clock for 128-bit key and 256-bit key sizes. Our core achieves a throughput of 0.38 Mbps at 1 MHz which is less than [79]. Since [79] is a commercial core, it may be highly optimized for that specific ASIC technology. Ours is more generalized and the results are from older technology.

Table 9.4: Implementation results using SAED_90nm ASIC library

Enc/Dec	Key size	mode	Latency	GE	F_{max}	Mbps	Kbps/GE
Encryption	128	ECB	339	6952	309.60	11.690	1.682
		CBC	339	6952	309.60	11.690	1.682
		CFB	323	6952	309.60	12.269	1.765
		CTR	323	6952	309.60	12.269	1.765
		OFB	323	6952	309.60	12.269	1.765
	256	ECB	467	6952	309.60	8.486	1.221
		CBC	467	6952	309.60	8.486	1.221
		CFB	451	6952	309.60	8.787	1.264
		CTR	451	6952	309.60	8.787	1.264
		OFB	451	6952	309.60	8.787	1.264
Decryption	128	ECB	348	6952	309.60	11.387	1.638
		CBC	348	6952	309.60	11.387	1.638
		CFB	323	6952	309.60	12.269	1.765
		CTR	323	6952	309.60	12.269	1.765
		OFB	323	6952	309.60	12.269	1.765
	256	ECB	480	6952	309.60	8.256	1.188
		CBC	480	6952	309.60	8.256	1.188
		CFB	451	6952	309.60	8.787	1.264
		CTR	451	6952	309.60	8.787	1.264
		OFB	451	6952	309.60	8.787	1.264

Chapter 10: Conclusion and Future Work

In this dissertation, we made efforts to reduce the complexity and time in designing lightweight architectures. We present a generalized methodology for developing lightweight architectures which helps in making various design choices such as interface, width of datapath, choice of processing elements etc. We developed a tool for optimizing control logic using memories and evaluate their effectiveness using AES and Keccak core.

Furthermore, using proposed methodology and optimization techniques, we present lightweight architectures of AES with three different datapath widths, SHA-256, KETJE-SR, ASCON-128, ASCON-128A. Multiple cryptographic services built on the same cryptographic primitives is attractive for resource constraints devices. We investigated this with AES and Keccak as the underlying primitive.

We developed lightweight CAESAR hardware LWAPI package for lightweight application and show its benefits using KETJE-SR and ASCON. Furthermore, we also developed lightweight architecture of AES targeted for ASICs which supports block cipher modes ECB, CBC, CFB, CTR, and OFB for 128 and 256-bit key lengths.

For future work, we want to extend our control logic optimization tool for asymmetric cryptographic algorithms and for non cryptographic algorithms. Power and energy are two important metrics for evaluation of lightweight designs. We would like to analyzing our lightweight architecture with respect to power and energy. Exploring various side channel protection techniques for lightweight architectures would be another good extension of this work.

Bibliography

[1] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems–CHES 2007*, ser. Lecture Notes in Computer Science (LNCS), vol. 4727. Springer, 2007, pp. 450–466.

[2] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee, "HIGHT: A new block cipher suitable for low-resource device," in *CHES 2006*, ser. LNCS, L. Goubin and M. Matsui, Eds., vol. 4249. Springer, 2006, pp. 46–59.

[3] D. Wheeler and R. Needham, "TEA extensions," Cambridge University, England, Tech. Rep., Oct 1997.

[4] S. Ojha, N. Kumar, K. Jain, and Sangeeta, "TWIS:a lightweight block cipher," in *Information Systems Security*, ser. Lecture Notes in Computer Science, A. Prakash and I. Sen Gupta, Eds. Springer Berlin / Heidelberg, 2009, pp. 280–291.

[5] *Data Encryption Standard (DES)*, National Institute of Standards and Technology (NIST), FIPS Publication 46-3, Oct 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[6] M. Izadi, B. Sadeghiyan, S. Sadeghian, and H. Khanooki, "MIBS: A new lightweight block cipher," in *Cryptology and Network Security*, ser. Lecture Notes in Computer Science, J. Garay, A. Miyaji, and A. Otsuka, Eds., vol. 5888. Springer Berlin / Heidelberg, 2009, pp. 334–348.

[7] "sourcetech411," http://sourcetech411.com/2013/04/top-fpga-companies-for-2013/, accessed:01-05-2015.

[8] "Microsemi," http://www.microsemi.com/products/fpga-soc/low-power, accessed:01-05-2015.

[9] L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic power consumption in virtex™-ii fpga family," in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, ser. FPGA '02. New York, NY, USA: ACM, 2002, pp. 157–164. [Online]. Available: http://doi.acm.org/10.1145/503048.503072

[10] A. Amara, F. Amiel, and T. Ea, "FPGA vs. ASIC for low power applications," *Microelectronics Journal*, vol. 37, no. 8, pp. 669–677, 2006.

[11] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal, The*, vol. 34, no. 5, pp. 1045–1079, Sept 1955.

[12] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton, NJ: Princeton University Press, 1956, pp. 129–153.

[13] M. Rawski, H. Selvaraj, and T. Luba, "An application of functional decomposition in ROM-based FSM implementation in FPGA devices," *J. Syst. Archit.*, vol. 51, no. 6-7, pp. 424–434, 2005.

[14] V. Skylarov, "Synthesis and implementation of RAM-based finite state machines in FPGAs," in *Field-Programmable Logic and Applications – FPL'00*, ser. LNCS, R. W. Hartenstein and H. Grünbacher, Eds., vol. 1896. Springer-Verlag, 2000, pp. 718–728.

[15] A. Tiwari and K. A. Tomko, "Saving power by mapping finite-state machines into embedded memory blocks in FPGAs," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, 2004, p. 20916.

[16] I. García-Vargas, R. Senhadji-Navarro, G. Jiménez-Moreno, A. Civit-Balcells, and P. Guerra-Gutiérrez, "ROM-based finite state machine implementation in low cost FPGAs," in *International Symposium on Industrial Electronics, ISIE 2007*. IEEE, June 2007, pp. 2342–2347.

[17] J.-P. Kaps, G. Gaubatz, and B. Sunar, "Cryptography on a Speck of Dust," *Computer*, vol. 40, no. 2, pp. 38–44, Feb 2007.

[18] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A Survey of Lightweight-Cryptography Implementations," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 522–533, Nov.-Dec. 2007.

[19] M. Kneevi, V. Nikov, and P. Rombouts, "Low-Latency Encryption Is Lightweight = Light + Wait?" in *Cryptographic Hardware and Embedded Systems CHES 2012*, ser. Lecture Notes in Computer Science, E. Prouff and P. Schaumont, Eds. Springer Berlin Heidelberg, 2012, vol. 7428, p. 426446. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33027-8_25

[20] S. Kerckhof, F. Durvaux, C. Hocquet, D. Bol, and F.-X. Standaert, "Towards green cryptography: A comparison of lightweight ciphers from the energy viewpoint," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 390–407. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33027-8_23

[21] P. Yalla and J.-P. Kaps, "Compact FPGA implementation of Camellia," in *Field Programmable Logic and Applications, FPL 2009*, M. Daněk, J. Kadlec, and B. Nelson, Eds. IEEE, Aug. 2009, pp. 658–661.

[22] ——, "Lightweight cryptography for FPGAs," in *International Conference on ReConFigurable Computing and FPGAs – ReConFig'09*. IEEE, Dec. 2009, pp. 225–230.

[23] J.-P. Kaps and B. Sunar, "Energy Comparison of AES and SHA-1 for Ubiquitous Computing," in *EUC-06*, ser. LNCS, vol. 4097. Springer, Aug 2006, pp. 372–381.

[24] T. Good and M. Benaissa, "AES on FPGA from the Fastest to the Smallest." in *CHES 2005*, ser. LNCS, J. R. Rao and B. Sunar, Eds., vol. 3659. Springer, 2005, pp. 427–440.

[25] P. Chodowiec and K. Gaj, "Very Compact FPGA Implementation of the AES Algorithm," in *CHES 2003*, ser. LNCS, vol. 2779. Springer, Sep. 2003, pp. 319–333.

[26] J.-P. Kaps, "Chai-tea, Cryptographic Hardware Implementations of xTEA," in *INDOCRYPT 2008*, ser. LNCS, D. Chowdhury, V. Rijmen, and A. Das, Eds., vol. 5365. Springer, Dec 2008, pp. 363–375.

[27] D. Hwang *et al.*, "Comparison of FPGA-Targeted Hardware Implementations of eSTREAM Stream Cipher Candidates," in *SASC Workshop 2008*, ser. , Feb 2008, pp. 151–162.

[28] J.-P. Kaps, P. Yalla, K. K. Surapathi, B. Habib, S. Vadlamudi, S. Gurung, and J. Pham, "Lightweight implementations of SHA-3 candidates on FPGAs," in *Progress in Cryptology – INDOCRYPT 2011*, ser. Lecture Notes in Computer Science (LNCS), D. J. Bernstein and S. Chatterjee, Eds., vol. 7107. Springer Berlin / Heidelberg, Dec 2011, pp. 270–289.

[29] J.-P. Kaps, P. Yalla, K. K. Surapathi, B. Habib, S. Vadlamudi, and S. Gurung, "Lightweight implementations of SHA-3 finalists on FPGAs," Washington, D.C., Mar 2012, third SHA-3 candidate conference.

[30] B. Jungk and J. Apfelbeck, "Area-efficient FPGA implementations of the SHA-3 finalists," in *International Conference on ReConfigurable Computing and FPGAs*. IEEE: ReConfig'11, DEC 2011.

[31] S. Banik, A. Bogdanov, and F. Regazzoni, "Exploring the energy consumption of lightweight blockciphers in fpga," *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, 2015.

[32] E. Homsirikamol, M. Rogawski, and K. Gaj, "Throughput vs area trade-offs architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs," in *CHES*, ser. LNCS, B. Preneel and T. Takagi, Eds., vol. 6917. Springer, Sep 2011, pp. 491–506.

[33] *Spartan-3 Generation, FPGA User Guide*, Ug331 (v1.2) ed., Xilinx, Inc., Apr 2007.

[34] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-bit block cipher suitable for multiple platforms – design and analysis," in *Selected Areas in Cryptography, SAC 2000*, ser. Lecture Notes in Computer Science (LNCS), vol. 2012. Springer, 2001, pp. 39–56.

[35] N. Nalla Anandakumar, T. Peyrin, and A. Poschmann, *A Very Compact FPGA Implementation of LED and PHOTON*. Cham: Springer International Publishing, 2014, pp. 304–321. [Online]. Available: https://doi.org/10.1007/978-3-319-13039-2_18

[36] B. Calhoun, F. Honore, and A. Chandrakasan, "Design methodology for fine-grained leakage control in mtcmos," in *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, Aug 2003, pp. 104–109.

[37] A. Gayasen, Y. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and T. Tuan, "Reducing leakage energy in fpgas using region-constrained placement," in *in Proc. ACM Intl. Symp. Field-Programmable Gate Arrays*, 2004, pp. 51–58.

[38] A. Rahman and V. Polavarapuv, "Evaluation of low-leakage design techniques for field programmable gate arrays," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '04. New York, NY, USA: ACM, 2004, pp. 23–30. [Online]. Available: http://doi.acm.org/10.1145/968280.968285

[39] E. Kusse and J. Rabaey, "Low-energy embedded fpga structures," in *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, Aug 1998, pp. 155–160.

[40] V. George, H. Zhang, and J. Rabaey, "The design of a low energy fpga," in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, Aug 1999, pp. 188–193.

[41] J. Rabaey and M. Pedram, *Low Power Design Methodologies*, ser. . Norwell, Massachusetts: Kluwer Academic Publishers, 1996.

[42] N. Grover and M. Soni, "Reduction of power consumption in FPGAs-an overview," *I.J. Information Engineering and Electronic Business*, pp. 50–69, Oct 2012. [Online]. Available: http://www.mecs-press.org/ijieeb/ijieeb-v4-n5/IJIEEB-V4-N5-7.pdf

[43] J. A. Brzozowski and T. Luba, "Decomposition of boolean functions specified by cubes," Journal of Multiple-Valued Logic and Soft Computing, Tech. Rep., 1997.

[44] *Advanced Encryption Standard (AES)*, National Institute of Standards and Technology (NIST), FIPS Publication 197, Nov 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[45] M. U. Sharif, R. Shahid, M. Rogawski, and K. Gaj, "Use of embedded FPGA resources in implementations of five round three SHA-3 candidates," ECRYPT II Hash Workshop 2011, May 2011.

[46] *Hardware Interface of a Secure Hash Algorithm (SHA)*, v. 1.4 ed., Cryptographic Engineering Research Group, George Mason University, Jan 2010.

[47] K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and comprehensive methodology for comparing hardware performance of fourteen round two SHA-3 candidates using FPGA," in *CHES*, ser. LNCS, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer, 2010, pp. 264–278.

[48] M. Dworkin, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication*, NIST, SP 800-38D, Apr 2006, draft.

[49] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, P. Yalla, J.-P. Kaps, and K. Gaj, "CAESAR hardware API," Cryptology ePrint Archive, Report 2016/626, 2016, http://eprint.iacr.org/2016/626.

[50] "Implementer's guide to hardware implementations compliant with the CAESAR hardware API version 2.0," https://cryptography.gmu.edu/athena/CAESAR_HW_API/caesar_hw_devpkg-2.0.zip, accessed: 2017-10-20.

[51] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "CAESAR submission:Ketje v2," Submission to CAESAR (Round3), September 2016, https://competitions.cr.yp.to/round3/ketjev2.pdf.

[52] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak reference," http://keccak.noekeon.org/Keccak-reference-3.0.pdf, Jan 2011.

[53] *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, National Institute of Standards and Technology (NIST), FIPS Publication 202, Aug 2015, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[54] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Duplexing the sponge: Single-pass authenticated encryption and other applications," in *SAC*, ser. LNCS, vol. 7118. Springer, 2012, pp. 320–337.

[55] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "CAESAR submission:Ketje v1," Submission to CAESAR (Round2), March 2014, https://competitions.cr.yp.to/round1/ketjev1.pdf.

[56] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "ASCON v1.2," Submission to CAESAR (Round3), September 2016.

[57] *Spartan-6 FPGA Configuration User Guide*, UG380 (v2.9) ed., Xilinx, Inc., Aug 2016, https://www.xilinx.com/support/documentation/user_guides/ug380.pdf.

[58] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa – automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs," in *FPL*. IEEE, 2010, pp. 414–421.

[59] "ATHENa database of FPGA results for authenticated ciphers," https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/table_view, accessed: 2017-10-20.

[60] S. Frankel, K. Kent, R. Lewkowski, A. D. Oerbaugh, R. W. Ritchey, and S. S. R., *Guide to IPsec VPNs*, NIST, SP 800-77, Dec 2005.

[61] A. Freier, P. Karlton, and P. Kocher, "The secure sockets layer (SSL) protocol," IETF, RFC 6101, Aug 2011.

[62] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," Network Working Group, RFC 5246, Aug 2008.

[63] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The Keccak SHA-3 submission," Submission to NIST (Round 3), 2011, http://keccak.noekeon.org/Keccak-submission-3.pdf.

[64] ——, "Cryptographic sponge function," http://sponge.noekeon.org/CSF-0.1.pdf, Jan 2011.

[65] F. K. Gürkaynak, K. Gaj, B. Muheim, E. Homsirikamol, C. Keller, M. Rogawski, H. Kaeslin, and J.-P. Kaps, "Lessons learned from designing a 65nm ASIC for evaluating third round SHA-3 candidates," Mar 2012, third SHA-3 Candidate Conference.

[66] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely, "High-speed hardware implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein," Cryptology ePrint Archive, Report 2009/510, Nov 2009.

[67] X. Guo, S. Huang, L. Nazhandali, and P. Schaumont, "Fair and comprehensive performance evaluation of 14 second round SHA-3 ASIC implementations," 2010, second SHA-3 Candidate Conference.

[68] L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller, and F. Gürkaynak, "Developing a hardware evaluation method for SHA-3 candidates," in *CHES*, ser. LNCS, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer, 2010, pp. 248–263.

[69] B. Cohen and B. Laurie, *AES-Hash*, Submission to NIST, May 2001, http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/aes-hash/aeshash.pdf.

[70] S. Matyas, C. Meyer, and J. Oseas, "Generating strong one-way functions with cryptographic algorithm," IBM Tech. Disclosure Bulletin, 5658–5659, Tech. Rep. 27, 1985.

[71] M. Dworkin, *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*, NIST, Special Publication 800-38B, Mar 2005.

[72] R. McEvoy, J. Curran, P. Cotter, and C. Murphy, "Fortuna:cryptographically secure pseudo-random number generation in software and hardware," in *Irish Signals and Systems Conference, 2006 IET*, June 2006, pp. 457–462.

[73] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. V. Keer, "Caesar submission: Keyak v1," http://competitions.cr.yp.to/round1/keyakv1.pdf, Mar 2014.

[74] P. Yalla, E. Homsirikamol, and J.-P. Kaps, "Comparison of multi-purpose cores of Keccak and AES," in *Design, Automation Test in Europe DATE 2015*. ACM, Mar 2015, pp. 585–588.

[75] Y. Lu, G. Shou, Y. Hu, and Z. Guo, "The research and efficient fpga implementation of ghash core for gmac," in *E-Business and Information System Security, 2009. EBISS '09*, 2009, pp. 1–5.

[76] *AES-GCM Core family for Xilinx FPGA*, Helion Technology, Fulbourn, Cambridge CB21 5DQ, England, 2011, http://www.heliontech.com/downloads/aes_gcm_8bit_xilinx_datasheet.pdf.

[77] *AES-CCM Core family for Xilinx FPGA*, Helion Technology Limited, Fulbourn, Cambridge CB21 5DQ, England, 2011, http://www.heliontech.com/downloads/Helion_PB_-_AES-CCM_8-bit_FPGA.pdf.

[78] M. Rogawski, "Development and Benchmarking of New Hardware Architectures for Emerging Cryptographic Transformations," Ph.D. dissertation, George Mason University, July 2013.

[79] *AES IP Core for ASICS*, Helion Technology, Fulbourn, Cambridge CB21 5DQ, England, 2014, http://www.heliontech.com/downloads/Helion_PB_-_AES_ASIC.pdf#view=Fit.

Curriculum Vitae

Panasayya Yalla received his Bachelor of Engineering from Andhra University, India in 2006. He graduated with Master of Science in Computer Engineering from George Mason University, USA, in 2009. During his Master's and Ph.D studies, he served as a research assistant developing several lightweight architectures for cryptographic applications. He also served as teaching assistant for several undergraduate and graduate courses.

Publications:

1. Panasayya Yalla and Jens-Peter Kaps. Compact FPGA implementation of Camellia, Field Programmable Logic and Applications, FPL 2009. In Martin Daněk, Jiří Kadlec, and Brent Nelson editors, IEEE, pages 658–661, Aug., 2009

2. Panasayya Yalla and Jens-Peter Kaps. Lightweight cryptography for FPGAs, International Conference on ReConFigurable Computing and FPGAs ReConFig'09, IEEE, pages 225–230, Dec., 2009

3. Jens-Peter Kaps, Panasayya Yalla, Kishore Kumar Surapathi, Bilal Habib, Susheel Vadlamudi, Smriti Gurung, and John Pham. Lightweight implementations of SHA-3 candidates on FPGAs, Progress in Cryptology INDOCRYPT 2011. In Daniel J. Bernstein and Sanjit Chatterjee editors, LNCS, volume 7107, Springer, pages 270–289, Dec., 2011

4. Jens-Peter Kaps, Panasayya Yalla, Kishore Kumar Surapathi, Bilal Habib, Susheel Vadlamudi, and Smriti Gurung. Lightweight implementations of SHA-3 finalists on FPGAs, *third* SHA-3 conference, Washington, D.C., Mar, 2012

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

6. William Diehl, Farnoud Farahmand, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. Comparison of hardware and software implementations of selected lightweight block ciphers, 27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, Sep., 2017

7. Panasayya Yalla and Jens-Peter Kaps. Evaluation of CAESAR hardware API for lightweight implementations, International Conference on Reconfigurable Computing and FPGAs (ReConFig 2017), Cancun, Mexico, Dec., 2017

8. Ahmad Salman, Ahmed Ferozpuri, Ekawat Homsirikamol, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. A scalable ECC processor implementation for high-speed and lightweight with side-channel countermeasures, International Conference on Reconfigurable Computing and FPGAs (ReConFig 2017), Cancun, Mexico, Dec., 2017

9. Jasper Van Woudenberg, Cees-Bart Breunesse, Rajesh Velegalati, Panasayya Yalla and Sergio Gonzalez. Differential Fault Analysis Using Symbolic Execution, 7th Software Security, Protection, and Reverse Engineering Workshop, Orlando, USA, Dec., 2017