TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

A Hardware - Accelerated Cryptography IP for Disaggregated Datacenters

---

Author:
Eleni DRAKOULAKI

Thesis Committee:
Prof. Apostolos DOLLAS
Assoc. Prof. Sotirios IOANNIDIS
Prof. Dionisios PNEVMATIKATOS (NTUA)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

7 July 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

A Hardware - Accelerated Cryptography IP for Disaggregated Datacenters

by Eleni DRAKOULAKI

The world generates an unfathomable amount of data every minute of every day, and it continues to multiply at a staggering rate. Organizations in every industry are rapidly moving from batch processing to real-time data streams to keep pace with modern demands. The need to secure and protect private and personal data is greater than ever, and the field of cryptography provides the tools to handle this task. This thesis presents a design for a hardware-accelerated cryptography IP with the goal to be incorporated into a disaggregated datacenter and protect the data without interfering with the bandwidth and latency requirements of the server. In this thesis, we present the implementation of the AES, RC6, and Blowfish algorithms, both encryption and decryption, in the Zynq UltraScale+ ZCU102 Evaluation Platform and their evaluation based on a series of simulation-level tests, with AES and RC6 achieving the best throughput at 12.79 Gbps, while AES kept the resource utilization at a low level. As well as, the evaluation of AES encryption and decryption design in a physical board, by using the PYNQ Z1 FPGA board, and its overall performance compared to a software implementation running in a high-speed server, and even though it loses in terms of performance against the Zynq UltraScale+ ZCU102, it proves that the design, while implemented at a small, low cost, low-power consumption FPGA board can perform as well in the case of encryption, or even two times better, in the case of decryption, against a high-speed server.

Acknowledgements

First of all, I would like to express my deepest appreciation to my supervisor Prof. Apostolos Dollas for his guidance, advice, and support during the course of this thesis. Additionally, I would like to thank Prof. Dionisios Pnevmatikatos for initiating this project and trusting me with it. I would like to express to both of them, how grateful I am for the knowledge and the opportunity they provided me to work in the amazing field of hardware.

Also, I would like to express my deepest gratitude to my two advisors, Dr. Dimitrios Theodoropoulos and Dr. Grigorios Chrysos. I would like to thank them for their help, guidance, and support in this project, as well as the patience they showed me. They have been great mentors for me during this journey and I consider myself extremely lucky to have had the opportunity to work by their side.

Furthermore, I would like to thank my thesis committee, Prof. Sotirios Ioannidis, for evaluating my work.

Also, I am grateful to my family for their love and support all of these years. I am also thankful to all my friends that I made throughout these years for the amazing moments we had, and even though we don't see each other very often, I know we will always be close to each other by heart.

Last but not least, I would like to thank my significant other Kostas, who has always been there for me, expressing his love and support, as well as his belief in my ability to accomplish anything I set my mind to.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The world generates an unfathomable amount of data every minute of every day, and it continues to multiply at a staggering rate. Organizations in every industry are rapidly moving from batch processing to real-time data streams to keep pace with modern demands.

The need to secure and protect private and personal data by using unique codes that encrypt the data and make it impossible for intruders to read is greater than ever.

Due to the nature of the data, we must keep in mind that the encryption/decryption process must be done as quickly as possible. No one has the time to wait for their data to be encrypted/decrypted.

1.1 Motivation

Cryptography is the study of secure communication techniques that allow only the sender and intended recipient of a message to view its contents. The term is derived from the Greek word "kryptos", which means hidden. It is closely associated to encryption, which is the act of scrambling ordinary text into what's known as ciphertext and then back again upon arrival. In addition, cryptography also covers the obfuscation of information in images using techniques such as microdots or merging. Ancient Egyptians were known to use these methods in complex hieroglyphics, and Roman Emperor Julius Caesar is credited with using one of the first modern ciphers.

The two main types of cryptography are the symmetric or "secret key" algorithms and the asymmetric or "public key" algorithms. In the first case, data is encrypted using a secret key, and then both the encoded message and secret key are sent to the recipient for decryption. In the other case, every user

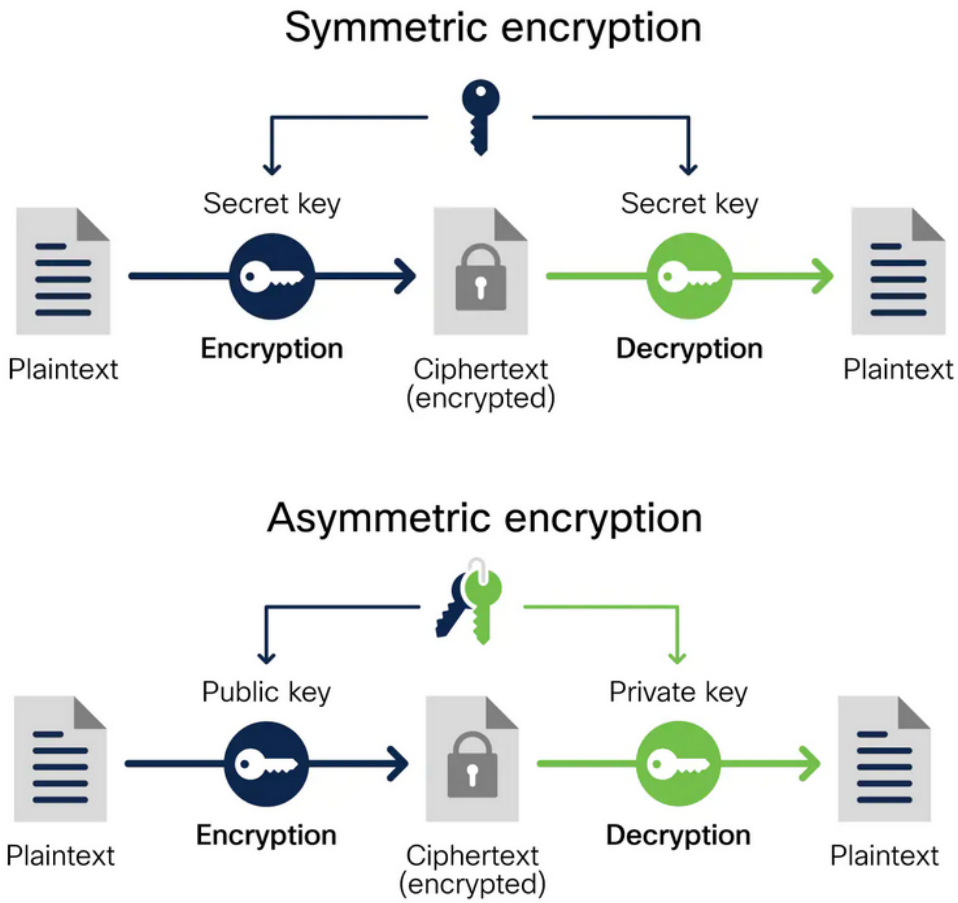


FIGURE 1.1: Symmetric and Asymmetric Key Algorithms
URL

has two keys: one public and one private. Senders request the public key of their intended recipient, encrypt the message and send it along. When the message arrives, only the recipient's private key will decode it.

In today's cloud datacenters, the physical system is composed of individual server units contributing processing, memory, accelerators, and storage resources. However, this arrangement incurs a significant waste of these resources as well as low power utilization, due to the inherent inability to closely match user IT requirements to the resources available within a single server or sets of servers. The challenge in this arrangement is to be more efficient, flexible, and agile with these resources. Disaggregated datacenters, such as dReDBox [1] aspires to remedy this by moving from today's server-as-the-unit model to a pooled-computing model, enabling an arbitrary sizing of disaggregated IT resources, deploying them where and when required, to perfectly match cloud user requirements.

FPGA programming has been gaining momentum lately, as it offers considerable benefits. It allows you to offload resource-hungry tasks to hardware and thus increase performance. FPGAs can be programmed and reprogrammed according to the needs of each application, which is very cost-effective in the long run.

Although specialized hardware has the potential to provide huge acceleration at a fraction of a processor's energy, the main drawback is related to its design. On the one hand, describing these components in a hardware description language (HDL) (e.g., VHSIC hardware description language (VHDL) or Verilog) allows the designer to incorporate existing tools for register transfer level (RTL) and logic synthesis into the target technology. On the other hand, this requires the developer to specify functionality at a low level of abstraction where cycle-by-cycle behavior is fully specified. Using such languages requires advanced hardware knowledge and is also complicated to develop. This leads to longer development times, which can have a critical impact on the time-to-market. An interesting solution to resolve this issue is to combine reconfigurable hardware architectures, such as field-programmable gate arrays (FPGAs) and high-level synthesis (HLS) tools. [2]

HLS tools start from a software programmable high-level language (HLL) (e.g., C, C++, and SystemC) to automatically create a circuit specification in HDL that performs the same function. HLS benefits software engineers by allowing them to profit by the speed and power advantages of hardware without having to build up hardware expertise, as well as design systems faster at a high-level of abstraction.

1.2 Scientific Contributions

The goal of this thesis was to explore and determine which of the algorithms belonging to the symmetric key encryption family when implemented in the ZCU102 Evaluation Platform by the use of HLS tools, have the potential to be incorporated into a disaggregated datacenter such as dRedBox, and effectively protect the data while supporting the bandwidth and latency requirements of the communication that is currently inside the server. This translates to a desirable throughput of at least 10 Gbps while keeping resources and initial latency at a minimal level.

The algorithms chosen for this task, based on their performance in existing hardware implementations, are the Advanced Encryption Standard (AES), RC6 (Rivest Cipher 6), and Blowfish.

The most significant contributions of our work are the following:

- Study of related works that map encryption algorithms on hardware-based platforms and concentrate their results.
- Map symmetric key encryption algorithms, i.e. the Advanced Encryption Standard (AES), RC6 (Rivest Cipher 6), and Blowfish, on reconfigurable platform and compare their architectural characteristics.
- Optimizing reconfigurable algorithm architectures and minimizing resources and latency when processing data streams of varying sizes.
- The proposed systems achieve high processing throughput rates up to 12.8 Gbps for AES and RC6 (for both encryption and decryption).
- Successfully download the design of a small, low-cost, low-energy consumption FPGA board, such as PYNQ-Z1.
- The design on the PYNQ-Z1 board can perform as well as a high-speed server in the case of AES Encryption, and 2x better in the case of AES Decryption.

1.3 Thesis Outline

- **Chapter 2 - Related Work:** Comparative study of hardware implementations of Symmetric-key algorithms.
- **Chapter 3 - Encryption/Decryption Algorithms:** Detailed presentation of the theoretical background for AES, RC6, and Blowfish algorithms.
- **Chapter 4 - Architecture and Detailed Design of AES Encryption:** Description of the architecture and detailed hardware implementation design of the AES Encryption algorithm and optimizations.
- **Chapter 5 -Hardware Architecture of AES Decryption, RC6 Encryption, RC6 Decryption, Blowfish Encryption, and Blowfish Decryption:** Description of the Hardware Architecture of all the rest algorithms.

- **Chapter 6 - Design Verification and Performance Evaluation from Actual Runs:** Description of the results and performance from the different platforms.
- **Chapter 7 - Conclusions and Future Work:** Conclusions of the thesis and ideas for further research.

Chapter 2

Related Work

Symmetric-key algorithms also known as single-key, one-key, and private-key encryption are a class of algorithms for cryptography, that uses a Private (shared secret) key and a Public (non-secret) algorithm to execute encryption /decryption process. The keys may be identical or there may be a simple transformation to go between the two keys. The keys, in practice, represent a shared secret between two or more parties that can be used to maintain a private information link.

In this chapter, we have concentrated various information on 16 symmetric encryption algorithms based on their performance in hardware implementations. For most of those algorithms, the scientific community has provided several publications with interesting results. We mostly focused on recent publications as possible with the use of Field-Programmable Gate Array (FPGA) devices and a targeted throughput of 10 Gbps and above.

The algorithms that we have researched are DES (Data Encryption Standard), TDES (or 3DES / TDEA - Triple Data Encryption Standard / Algorithm), AES (Advanced Encryption Standard, Rijndael), Blowfish, Serpent, Twofish, RC6 (Rivest Cipher 6), MARS, IDEA (International Data Encryption Algorithm), TEA (Tiny Encryption Algorithm), XTEA (eXtended TEA), CAST-128 (alternatively CAST5), MISTY1, KHAZAD, ARIA, and Camellia. [3] [4] [5]

Our goal is to choose three algorithms that can achieve the desired throughput and also provide a good level of security with low resource utilization and low latency.

Algorithms / Parameters		DES			3DES (TDES, TDEA)			Blowfish	
Published		1977			1998			1993	
Developed by		IBM			IBM			Bruce Schneier	
Algorithms Structure		Feistel			Feistel			Feistel	
Key Length		56 bits (+8 parity bits)			112 bits, 168 bits			32 – 448 bits	
Flexibility or Modification		No			Yes, Extended from 56 to 168 bits			YES,64-448 key size in multiple of 32	
Number of Rounds		16			48			16	
Block size		64 bits			64 bits			64 bits	
Attacks		Brute force attack			Brute force attack, Known plaintext, Chosen plaintext			Dictionary attack	
Author		S.Oukili, S. Bri [6] (2015)	C.Patterson [7] (2000)	Vikram Pasham and Steve Trimberger [8] (2001)	P. Kitsos, N. Sklavos, M.D. Galanis, O. Koufopavlou [9] (2004)	Vikram Pasham and Steve Trimberger [8] (2001)	E.J. Swankoski, R.R. Brooks, V. Narayanan, M. Kandemir, M.J. Irwin [10] (2004)	S.Oukili, S. Bri [15] (2016)	Nalawade, Gawali [16] (2017)
Device used		Spartan-3e XC3S500E	Virtex XCV150	Virtex-II XC2V1000-5FG456	Virtex XCV1600EBG560-6	Virtex-II XC2V3000-5FG67	Virtex-II Pro	Virtex - 5 (xc5vlx220t-2ff1738)	Virtex-5 XC5VLX50T
Throughput (Gbps)		9.453	10.752	15.100	6.9	13.3	8.631	12.008	13.056 (= 1632 Mbytes/sec)
System Clock (MHz)		147.71	168	237	108	207	134.862	187.633	153
Resourses	CLB slices	2046 (43%)	1584 (91%)	--	14240	--	14525	1280 (3%)	--
	LUTs	--	2560	5036 (49%)	--	16,181 (56%)	--	3263 (2%)	939 (3%)
	Flip Flops	1143 (12%)	1248	5185 (50%)	--	15,759 (54%)	--	3002 (2%)	420 (1%)
	BRAM	--	--	--	--	--	--	79 (37%)	4 (6%)
Latency (clock cycles)		17 (then encrypts one data block (64 bits) per clock cycle)	--	48 cycles	48 cycles	144 cycles	0	34 clock cycles latency first time only. Then we recover the output at each clock cycle.	--

FIGURE 2.1: Comparative Analysis Table of Symmetric - Key Algorithms (Part 1)

Algorithms / Parameters		AES (Rijndael)					Serpent			
Published		2001					1998			
Developed by		Vincent Rijmen , Joan Daeman					Ross Anderson, Eli Biham, Lars Knudsen			
Algorithms Structure		Substitution - Permutation					Feistel			
Key Length		128 bits, 192 bits and 256					128 bits, 192 bits and 256			
Flexibility or Modification		YES, 256 key size is multiple of 64					Yes, Extended to 256 bits			
Number of Rounds		10, 12, 14					32			
Block size		128 bits					128 bits			
Attacks		Side channel attack					linear cryptanalysis, breaks 11 rounds of Serpent-128			
Author		Farooq, Aslam [11] (2016)		Farashahi, R.R., Rashidi, B., Sayedi, S.M. [12] (2014)	S.S.H. Shan, G. Raga [113] (2015)	S.Oukili, S. Bri [14] (2017)	Sugier Jarosław [17] (2012)	J. L´azaro, A. Astarloa, J. Arias, U. Bidarte, and C. Cuadrado [18] (2004)	B. Najafi, B. Sadeghian ,M. Saheb Zamani, A. Valizadeh [19] (2004)	Kris Gaj and Pawel Chodowiec [20] (2001)
Device used		Virtex-5	Spartan-6 (xc6s1x150-3-fgg900)	Virtex-5 XC5VLX8	Virtex-5 XC5VLX50	Virtex-6 (xc6vlx240t-3ff1156)	Spartan-6 XC6SLX75	Virtex-II X2C2000-6	Virtex XCV1000	Virtex XCV-1000BG560-6 (they used 2 devices)
Throughput (Gbps)		82.47	113.5	86	45	79	21.1	42.838	22	16.8
System Clock (MHz)		644.33	886.64	671.524	351.66	617.627	169	333	--	up to 200 MHz * (internal clock 131)
Resourses	CLB slices	--	--	--	--	4830 (12%)	--	8013	7504 (61%)	19,700
	LUTs	9276	9375	3557	4234	14736 (9%)	22029	--	--	--
	Flip Flops	255	256	2132	3256	18305 (6%)	16768	--	--	--
	BRAM	0	0	0	24	0	0	--	--	0
Latency (clock cycles)		--	--	--	--	80 clock cycles latency first time only. Then we recover the output at each clock cycle.	34	--	--	--

FIGURE 2.2: Comparative Analysis Table of Symmetric - Key Algorithms (Part 2)

Algorithms / Parameters		Twofish		RC6		MARS	IDEA	TEA
Published		1998		1998		1998	1990	1994
Developed by		Bruce Schneier		Ron Rivest, Matt Robshaw, Ray Sidney, Yiqun Lisa Yin		IBM	James L. Massey , Xuejia Lai	David Wheeler , Roger Needman
Algorithms Structure		Feistel		Feistel		Feistel	Substitution - Permutation	Feistel
Key Length		128 bits, 192 bits and 256		128 bits, 192 bits and 256		128 - 448 bits	128 bits	128 bits
Flexibility or Modification		Yes, Extended to 256 bits		Yes, 128-2048 key size is multiple to 32		Yes, 128 - 448 key size is multiple to 32	No	No
Number of Rounds		16		20		32	8	64 (32 cycles)
Block size		128 bits		128 bits		128 bits	64 bits	64 bits
Attacks		Differential attack, related key attack		Known plaintext		meet - in - the - middle attack can break 21 out of 32 rounds	key-schedule attacks, related-key differential timing attacks	related-key attack
Author		Kris Gaj and Pawel Chodowiec [20] (2001)	David Smekal, Jan Hajny, and Zdenek Martinasek [21] (2018)	Kris Gaj and Pawel Chodowiec [20] (2001)	J. L. Beuch [22] (2003)	Dimitris Theodoropoulos, Alexandros Siskos, Dionisis Pnevmatikatos [23] (2009)	Jean - Luc Beuchat [24] (2003)	M. A. Hussain and R. Badar [25] (2015)
Device used		Virtex XCV-1000BG560-6 (they used 2 devices)	Xilinx Virtex-7 HT (network card NFB-100G2Q)	Virtex XCV-1000BG560-6 (they used 4 devices)	Virtex-II XC2V3000-6	CCproc 4-core XC4VLX200	Virtex-II XC2V1000-6	Spartan 6-xc6slx45
Throughput (Gbps)		15.2	30.72	13.1	15.2	0.143	8.5	7.7
System Clock (MHz)		up to 200 MHz *	240	up to 200 MHz *	--	95 MHz	133.3	--
Resourses	CLB slices	21,000	--	46,900	8554 (59%)	--	3077(60%)	--
	LUTs	--	187070	--	--	--	--	--
	Flip Flops	--	91654	--	--	--	--	--
	BRAM	0	259	0	--	--	--	--
Latency (clock cycles)		--	--	--	--	338 cycles	--	--

FIGURE 2.3: Comparative Analysis Table of Symmetric - Key Algorithms (Part 3)

Algorithms / Parameters		XTEA (eXtended TEA)	CAST128 (CAST5)	MISTY1		Khazad	Camellia	ARIA
Published		1997	1996	1995		2000	2000	2003
Developed by		David Wheeler , Roger Needman	Carlisle Adams , Srafford Tavares	M.Matsui,T.Ichikawa,T.Sorimachi, T.Tokita,A. Yamagishi		Vincent Rijmen and Paulo S. L. M. Barreto	Mitsubishi Electric, NTT	group of South Korean researchers
Algorithms Structure		Feistel	Feistel	Feistel		substitution - permutation	Feistel	Substitution - permutation
Key Length		128 bits	40 - 128 bits	128 bits		128 bits	128, 192 or 256 bits	128, 192, or 256 bits
Flexibility or Modification		No	Yes, Extended to 64,128,256 bits	No		No	Yes	Yes
Number of Rounds		variable, recomended 64 rounds (32 cycles)	12 - 16	4×n (8 recommended)		8	18 or 24	12, 14, or 16
Block size		64 bits	64 bits	64 bits		64 bits	128 bits	128 bits
Attacks		related-key rectangle attack	differential related-key attack	integral cryptanalysis		No attack better than on the first five rounds is known	no known successful attacks	Meet-in-the-middle attack on 8 rounds
Author		Jens-Peter Kaps [26] (2008)	P. Kitsos, N. Sklavos, M.D. Galanis, O. Koufopavlou [9] (2004)	P. Kitsos, M.D. Galanis, and O. Koufopavlou [27] (2005)	G. Rouvroy, F.X. Standaert, J.J. Quisquater, J.D. Legat [28] (2003)	G. Rouvroy, F.X. Standaert, J.J. Quisquater, J.D. Legat [29] (2002)	Zoran Čiča [30] (2016)	Sang-Woo Lee, Sang-Jae Moon, and Jeong-Nyeo Kim [31] (2008)
Device used		Virtex 5 xc5vlx85-3	Virtex XCV1600EBG560-6	XCVII3000BF957-6	VIRTEXII2000bg575-6	VIRTEX1000BG560-6	Virtex5 XC5FX70T-1FF1136	0.25 µm CMOS technology
Throughput (Gbps)		20.645	3.39	12.6	10.1	9.472	32	43,338
System Clock (MHz)			53	168	159	148	251.2	338
Resourses	CLB slices	9647	24200	4039	6322	8800	Encryption 9753 (21%) Decryption 9753 (21%)	260,354 gates
	LUTs	--	--	--	11160	11328	--	--
	Flip Flops	--	--	--	11940	13568	Encryption 3369 (7%) Decryption 3352 (7%)	--
	BRAM	--	--	--	--	--	--	--
Latency (clock cycles)		192 clock cycles	--	--	208 clock cycles (output every 1 cycle)	62 cycles (output every 1 cycle)	24 clock cycles	--

FIGURE 2.4: Comparative Analysis Table of Symmetric - Key Algorithms (Part 4)

2.1 Symmetric Key Algorithms

2.1.1 DES - Data Encryption Standard

DES is the earliest symmetric encryption algorithm developed by IBM in 1972 and adopted in 1977 as the Federal Information Processing Standard (FIPS) by the National Bureau of Standards (NBS).

It includes 64 bits key that contains 56 bits that are directly utilized by the algorithm as key bits and are randomly generated and the remaining 8 bits, that are not used by the algorithm, are used for error detection as set to make a parity of each 8-bit byte. DES utilized the 56 bits key for the encryption and decryption process and performs the encryption of messages using the 64 bits block size. Similarly, the decryption process on a 64 bits ciphertext by using the same 56 bits key to produce the original 64 bits block of the message. The DES algorithm processes the 64 bits input with an initial permutation, 16 rounds of the key, and the final permutation. The DES algorithm structure is based on Feistel function F which divides the block into two halves. The function F is based on four stages such as expansion, key mixing, substitution, and permutation.

DES is mostly used in the banking industry, commercial and military secret information sharing purposes. Security is the major concern in DES because it uses the 56 bits key (256) or 7.2 x 1016 keys and cryptanalysts are trying to crack an encrypted message by key exhaustion. Brute force attack is possible through parallel machines of more than 2000 nodes with each node that has capabilities of key search 50 million keys/sec. DES was cracked in 1998 by using Electronic Freedom Foundation constructed device within 22 hours due to the less number of key length and is highly susceptible to the linear cryptanalysis attacks.[4]

S.Oukili, et. al. [6] in 2015 presented an efficient implementation of a 16-stage pipelined DES algorithm using time variable permutations. As a result, they increased the security of the algorithm achieving a throughput of 9.453 Gbps with 147.71MHz clock rate. Their design was implemented on a Spartan-3e device XC3s500e-4fg320. It occupied 2046 (43%) CLB slices, and 1143 (12%) slice Flip Flops and it takes 17 clock cycles latency first time only then encrypts one data block (64 bits) per clock cycle.

C. Patterson [7] in 2000 with a Java-base (Jbits) implementation achieved a fast encryption rate of 10.752 Gbps with 168 Mhz clock rate, using 1584 CLBs

(91%) of the Virtex XCV150. That makes it really impressive, considering the great result in such old technology.

V.Pasham, et. al. [8] in 2001 using loop unrolling and pipelining to gain speed, succeeded to achieve a throughput of 15.1 Gbps with 237 Mhz clock rate, using 5036 LUTs (49%) and 5185 Flip Flops (50%) on a Virtex-II XC2V1000 -5FG456 device, with an initial latency of 48 clock cycles.(16 copies of the round were built to unroll the loop, pipelining the data through the 16 stages, each round of DES is pipelined in three stages to enhance performance).

2.1.2 TDES - Triple Data Encryption Standard

Triple Data Encryption Standard (TDES or 3DES) referred as Triple Data Encryption Algorithm (TDEA) that was firstly proposed by IBM in 1998 and standardized in ANSI X9.17 and ISO 8732. TDES has appeared as the replacement of DES due to the improvement in the key length and applies the DES algorithm three times in each data block. The 56 bits key length of DES algorithm was generally adequate earlier when the algorithm was designed, but as the computation power increases then the brute force attack is feasible. On the other hand, TDES provides a very simple method by the increment of key length instead of designing a complete block cipher and it protects against the brute force attack. The key length for the TDES is 112 bits and 168 bits, the number of rounds is 48, and the block size is 64 bits. The purpose of this algorithm is to increase the security with longer key length, so it is challenging for the cryptanalyst to predict the pattern, and attacks become rapidly impractical. [4]

P. Kitsos, et. al. [9] in 2004 using a full loop unrolling architecture with a 48-stage pipeline achieved a throughput of 6.9 Gbps with 108MHz clock rate on a Virtex XCV1600EBG560-6 device. It occupied 14240 CLB slices with a 0.44 μs latency (around 48 clock cycles).

V.Pasham, et. al. [8] in 2001 using three copies of the DES implementation mentioned earlier achieved a throughput of 13.3 Gbps with a 207 MHz clock rate. Their design was implemented on a Virtex-II XC2V3000-5FG676 device. It occupied 16181 (56%) LUTs and 15759 (54%) slice Flip Flops. The latency for Triple DES implementation is 144 cycles, i.e., 48 cycles for each copy of DES.

E.J. Swankoski, et. al. [10] in 2004 proposed a parallel architecture in which internal hardware functionality is not duplicated but reused. This creates a

reasonably compact single block, which is ideal for duplication. This architecture includes 17 separate parallel DES or Triple DES blocks on a Virtex-II device (51 parallel DES Blocks). Given that the Triple DES implementation takes up 3.47% of the Virtex-II device and each single DES encryption or decryption operation has a latency of 17 cycles, each Triple DES Block could be duplicated 17 times to create a high-throughput zero latency cryptoprocessor. As a result, they achieved a throughput of 8.631 Gbps with a 134.862 MHz clock rate using 14525 CLB slices with zero latency.

2.1.3 AES - Advanced Encryption Standard

The NIST announced a call for the candidates of a cipher to implement a new encryption standard in 1997 because of the need for high security and efficiency, it's time to replace the existing DES and TDES encryption algorithm with new AES. All candidates of ciphers submitted their proposal by 1998 and finalized it in 2000. Finally, Rijndael was selected as the AES out of 15 candidates. Rijndael was developed by Vincent Rijmen and Joan Daemen in 2001. The US government is employed AES to protect sensitive information and implemented it across the world for data encryption purposes in the form of software and hardware.

AES appears as the recent generation block cipher and significantly increases in the block size up to 128 bits with key sizes 128 bits, 192 bits, and 256 bits. The number of rounds set with the respective key size is 10, 12, and 14 for the 128 bits, 192 bits, and 256 bits, respectively. The data blocks are used as the array of bytes and represented in a matrix that is referred to as the state array which changed in every step of the encryption and decryption process. Each round follows some steps during the encryption process to complete each round. The steps for each round consist of four layers i.e. substitute byte, shift rows, mix column and add round key. After the final step, the state array is transferred into the output matrix.

U. Farooq, et. al. [11] in 2016 presented a work that AES is implemented on FPGA using five different techniques. These techniques are based on optimized implementation of AES on FPGA by making efficient resource usage of the target device. Experimental results obtained are quite varying in nature. They range from smallest (suitable for area critical applications) to fastest (suitable for performance critical applications) implementation. The results of their best technique are 1) throughput of 113.5 Gbps with 886.64 MHz clock rate using 9375 LUTs, 256 flip flops, and 0 BRAMs on a Spartan-6

(xc6s1x150-3-fgg900) device, 2) throughput of 82.47 Gbps with 644.33 MHz clock rate using 9276 LUTs, 255 flip flops and 0 BRAMs on a Virtex - 5 FPGA device.

R.R. Farashahi, et. al. [12] in 2014 presented a high throughput digital design of the 128-bit Advanced Encryption Standard (AES) algorithm based on the 2-slow re-timing technique on FPGA. The C-slow re-timing is a well-known optimization and high-performance technique. It can enhance designs with feedback loops and automatically rebalances the registers in the design. The C-slow re-timing can break the critical path of the design into finer pieces to improve the throughput of the design. This work has been implemented using a Virtex - 5, XC5VLX85 FPGA device. It achieves a high throughput of 86 Gbps and a high maximum operation frequency of 671.524 MHz, using 3557 LUTs and 2132 Flip Flops.

S.S.H. Shan, et. al. [13] in 2015 described an FPGA implementation of chaotic-based advanced encryption standard (AES) using pipeline technique. The algorithm is a combination of chaotic maps and AES. In the proposed architecture, the AES key is generated by chaotic maps, and encryption is done by AES. The internal operations of each round of AES are optimized and parallel RAMs are used to implement the Sub-Bytes operation. They achieved a throughput of 45 Gbps with a 351.66 MHz clock rate on a Virtex-5 XC5VLX50 device. It occupied 3256 slice registers, 4234 LUTs, and 24 BRAMs.

S.Oukili, et. al. [14] in 2017 presented a high-speed efficient AES architecture. Pipelining technique is performed to obtain high throughput than the basic structure by inserting registers in optimum placements. Also, by employing a 5-stage pipelining S-box using combinational logic circuits to break the critical path delay, increased the speed and reduced the used resources. They implemented their design on a Virtex-6 device (xc6vlx240t-3ff1156). It occupied 4830 (12%) slices, 18305 (6%) slice registers, and 14736 (9%) slice LUTs. It takes 80 clock cycles latency for the first time only. Then, can recover the output at each clock cycle. The design achieves a maximum clock frequency of 617.627 MHz and a throughput of 79 Gbps.

2.1.4 Blowfish

The Blowfish algorithm is a 64-bit block cipher. It was designed in 1993 by Bruce Schneier as a symmetric block cipher. The Blowfish algorithm has a

variable key, from 32 bits to 448 bits. It has a 16-round Feistel function. Blowfish is unpatented and license-free and is available free for all users. It is effectively used for the encryption process and providing security to confidential data.

S.Oukili, et. al. [15] in 2016 presented a pipeline and parallel encryption techniques for the Blowfish algorithm. The pipelining technique modifies the critical path by increasing the possible frequency of the clock cycle. It consists in parallelizing the data inputs and outputs with the processing. Consequently, the algorithm is divided into stages, and registers are placed. As a result of this, the throughput can be increased. The FPGA implementation of the proposed Blowfish architecture was established on the Virtex-5 device (xc5vlx220t-2ff1738). They achieved a throughput of 12.008 Gbps at 187.633 MHz clock rate and used 1280 CLBs (3%), 3163 LUTs (2%), 3002 flip flops (2%), and 79 BRAMs (37%). The first encrypted data takes 34 clock cycles latency. Then, we recover the forthcoming encrypted data at each clock cycle.

S. B. Nalawade, et. al. [16] in 2017 presented a hardware implementation of the Blowfish algorithm. They achieved a throughput of 12.056 Gbps (1632 Mbps) at 153 MHz clock rate and used 939 LUTs (3%), 420 flip flops (1%), and 4 BRAMs (6%) of a Virtex-5 XC5VLX50T.

2.1.5 Serpent

Serpent is a symmetric key block cipher that was a finalist in the Advanced Encryption Standard (AES) contest, where it was ranked second to Rijndael. Serpent was designed by Ross Anderson, Eli Biham, and Lars Knudsen.

Like other AES submissions, Serpent has a block size of 128 bits and supports a key size of 128, 192, or 256 bits. The cipher is a 32-round substitution–permutation network operating on a block of four 32-bit words. Each round applies one of eight 4-bit to 4-bit S-boxes 32 times in parallel.

J. Sugier [17] in 2012 published the results of a hardware implementation of the Serpent cipher which in its fully pipelined version achieved a throughput of 21.1 Gbps on a Spartan-6 XC6SLX75 at 169 MHz clock rate and used 22029 LUTs(47.22%), 16768 flip flops (17.97%) and 0 BRAMs(0%) with an initial latency of 34 clock cycles.

J. Làzaro, et. al. [18] in 2004 presented a fully pipelined Serpent architecture implemented in a Virtex - II X2C2000 - 6 FPGA device, and runs at a

throughput of 42.8 Gbps. The encryption stage is fully pipelined. Through reconfiguration encryption and decryption share the same hardware structure. This achieved an FPGA area usage at 8.013 slices.

B. Najafi, et. al. [19] in 2004 presented a pipelined design on a Virtex XCV - 1000 FPGA which can encrypt/decrypt with 172.12 MHz clock rate that yields in 22 Gbps throughput.

K. Gaj, et. al. [20] in 2001 published a research of a hardware implementation for four AES candidates using a high-throughput architecture with pipelining inside and outside of cipher rounds, achieving speeds ranging over 12.2 Gbps. Specifically for Serpent by using two FPGA devices XCV-1000, achieved a throughput of 16.8 Gbps using 19700 slices of the devices.

2.1.6 Twofish

Twofish is a symmetric key block cipher with a block size of 128 bits and key sizes up to 256 bits. It was one of the five finalists of the Advanced Encryption Standard contest. Twofish is related to the earlier block cipher Blowfish. Twofish was designed by Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson.The Twofish cipher works with a 16-round iteration structure of the Feistel network, where function F is the basis.

Towfish was one of the algorithms that was implemented by K. Gaj, et. al. [20]. With the use of two XCV - 1000 FPGA devices, achieved a throught of 15.2 Gbps with area usage of 21000 slices.

D. Smekal, et. al. [21] in 2018 described in their article a hardware-accelerated implementation of the Twofish encryption algorithm on network card NFB-100G2Q equipped with a powerful FPGA chip Xilinx Virtex-7 HT. The encryption core was implemented to achieve real-time encryption and decryption. The algorithm was implemented for 128-bit words and 128-bit keys. This article demonstrates that the Twofish encryption core can operate with clock frequencies of 240 MHz and achieves a throughput of 30.72 Gbps. The design takes 45% of available Logic LUT and 6 % Memory LUT resources. Then it makes use of 13% FF registers and 28% of available RAM.

2.1.7 RC6 - Rivest cipher 6

RC6 (Rivest cipher 6) is a symmetric key block cipher derived from RC5. It was designed by Ron Rivest, Matt Robshaw, Ray Sidney, and Yiqun Lisa Yin to meet the requirements of the Advanced Encryption Standard (AES) competition. The algorithm was one of the five finalists and also was submitted to the NESSIE and CRYPTREC projects. It was a proprietary algorithm, patented by RSA Security.

RC6 proper has a block size of 128 bits and supports key sizes of 128, 192, and 256 bits up to 2040 bits, but, like RC5, it may be parameterized to support a wide variety of word lengths, key sizes, and number of rounds. RC6 is very similar to RC5 in structure, using data-dependent rotations, modular addition, and XOR operations. In fact, RC6 could be viewed as interweaving two parallel RC5 encryption processes, although RC6 does use an extra multiplication operation not present in RC5 in order to make the rotation dependent on every bit in a word, and not just the least significant few bits.

K. Gaj, et. al. [20] also implemented RC6 block cipher. In this case, by using four XCV-1000 FPGA devices, achieved a throughput of 13.1 Gbps with an area consumption of 46900 slices.

J.-L. Beuchat [22] in their paper, implemented and compared several implementations of the RC6 block cipher on Virtex - E and Virtex - II devices. They described several architectures of a RC6 processor designed for feedback or non-feedback chaining modes, and their fastest implementation achieved a throughput of 15.2 Gbps on a Xilinx XC2V3000-6 device with 8554 (59%) area utilization.

2.1.8 MARS

MARS is a block cipher that was IBM's submission to the Advanced Encryption Standard process. MARS was selected as an AES finalist in August 1999, after the AES2 conference in March 1999, where it was voted as the fifth and last finalist algorithm.

The MARS design team included Don Coppersmith, who had been involved in the creation of the previous Data Encryption Standard (DES) twenty years earlier. The project was specifically designed to resist future advances in cryptography by adopting a layered, compartmentalized approach.

MARS has a 128-bit block size and a variable key size of between 128 and 448 bits (in 32-bit increments). Unlike most block ciphers, MARS has a heterogeneous structure: several rounds of a cryptographic core are "jacketed" by unkeyed mixing rounds, together with key whitening.

D.Theodoropoulos, et. al. [23] in 2009 presented CCProc, a flexible cryptography coprocessor for symmetric-key algorithms. They designed an Instruction Set Architecture tailored to symmetric-key ciphers and built a hardware processor prototype. With a 4-core FPGA implementation mapped on XC4VLX200 FPGA at 95 MHz yielded a throughput of 143 Mbps (= 0.143 Gbps) for MARS block cipher.

2.1.9 IDEA

International Data Encryption Algorithm (IDEA), originally called Improved Proposed Encryption Standard (IPES), is a symmetric-key block cipher designed by James Massey of ETH Zurich and Xuejia Lai and was first described in 1991.

IDEA operates on 64-bit blocks using a 128-bit key and consists of a series of 8 identical transformations and an output transformation (the half-round). The processes for encryption and decryption are similar. IDEA derives much of its security by interleaving operations from different groups — modular addition and multiplication, and bitwise eXclusive OR (XOR) — which are algebraically "incompatible" in some sense.

J.-L. Beuchat [24] in their paper described several architectures of the IDEA block cipher and compared them from different points of view: throughput to area ratio or adequation with feedback and non-feedback chaining modes. Their fastest circuit achieved a throughput of 8.5 Gbps with a clock rate of 133.3 MHz on a XC2V1000 - 6 FPGA device and 3077 (60 %) area usage.

2.1.10 TEA and XTEA

2.1.10.1 TEA - Tiny Encryption Algorithm

Tiny Encryption Algorithm (TEA) is a block cipher notable for its simplicity of description and implementation, typically a few lines of code. It was designed by David Wheeler and Roger Needham of the Cambridge Computer Laboratory. It was first presented at the Fast Software Encryption workshop in Leuven in 1994.

TEA operates on two 32-bit unsigned integers (could be derived from a 64-bit data block) and uses a 128-bit key. It has a Feistel structure with a suggested 64 rounds, typically implemented in pairs termed cycles. It has an extremely simple key schedule, mixing all of the key material in exactly the same way for each cycle. Different multiples of a magic constant are used to prevent simple attacks based on the symmetry of the rounds. The magic constant, 2654435769 or 0x9E3779B9 is chosen to be $\lfloor 2^{32}/\phi \rfloor$, where ϕ is the golden ratio.

M.A. Hussain, et. al. [25] in 2015 presented hardware implementation of TEA block cipher on a Spartan 6 - xc6slx45 FPGA device for three different design approaches, sequential, combinational, and a hybrid of those two, achieving with the last approach a throughput of 7.7 Gbps.

2.1.10.2 XTEA - eXtended TEA

XTEA (eXtended TEA) is a block cipher designed to correct weaknesses in TEA. The cipher's designers were David Wheeler and Roger Needham of the Cambridge Computer Laboratory, and the algorithm was presented in an unpublished technical report in 1997.

Like TEA, XTEA is a 64-bit block Feistel cipher with a 128-bit key and a suggested 64 rounds. Several differences from TEA are apparent, including a somewhat more complex key-schedule and a rearrangement of the shifts, XORs, and additions. [3]

Jens-Peter Kaps [26] presented in their research an efficient implementation of XTEA on FPGAs and ASICs for ultra-low power applications such as RFID tags and wireless sensor nodes as well as fully pipelined designs for high-speed applications. A novel ultra-low power implementation was introduced which consumed less area and energy than a comparable AES implementation. The high-speed implementations of XTEA operated at 20.6 Gbps on a Virtex 5 xc5vlx85-3 FPGA device with area utilization of 9647 slices and 192 clock cycles latency.

2.1.11 CAST - 128

CAST-128 (alternatively CAST5) is a symmetric-key block cipher used in a number of products, notably as the default cipher in some versions of GPG and PGP. It has also been approved for the Government of Canada use by the

Communications Security Establishment. The algorithm was created in 1996 by Carlisle Adams and Stafford Tavares using the CAST design procedure.

CAST-128 is a 12- or 16-round Feistel network with a 64-bit block size and a key size of between 40 and 128 bits (but only in 8-bit increments). The full 16 rounds are used when the key size is longer than 80 bits.

Components include large 8×32-bit S-boxes based on bent functions, key-dependent rotations, modular addition and subtraction, and XOR operations. There are three alternating types of round function, but they are similar in structure and differ only in the choice of the exact operation (addition, subtraction, or XOR) at various points.[3]

P. Kitsos, et. al. [9] in 2004 using a full loop unrolling architecture achieved a throughput of 3.3 Gbps with 53 MHz clock rate on a Virtex XCV1600EBG560-6 device. It occupied 24200 CLB slices with a 0.3 μs latency.

2.1.12 MISTY1

MISTY1 (or MISTY-1) is a block cipher designed in 1995 by Mitsuru Matsui and others for Mitsubishi Electric. "MISTY" can stand for "Mitsubishi Improved Security Technology". It is also the initials of the researchers involved in its development: Matsui Mitsuru, Ichikawa Tetsuya, Sorimachi Toru, Tokita Toshio, and Yamagishi Atsuhiro.

MISTY1 is one of the selected algorithms in the European NESSIE project and has been among the cryptographic techniques recommended for Japanese government use by CRYPTREC in 2003. It was successfully broken in 2015 by Yosuke Todo using integral cryptanalysis.

MISTY1 is a Feistel network with a variable number of rounds (any multiple of 4), though 8 are recommended. The cipher operates on 64-bit blocks and has a key size of 128 bits. MISTY1 has an innovative recursive structure; the round function itself uses a 3-round Feistel network. MISTY1 claims to be provably secure against linear and differential cryptanalysis. [3]

P. Kitsos, et. al. [27] in 2005 presented a hardware architecture and an FPGA implementation of the MISTY1 block cipher. In their architecture, the MISTY1 rounds are unrolled and RAM blocks embedded in the XCVII3000 FPGA device are used for the implementation of the S-boxes. With 75-stage pipeline achieved a maximum throughput of 12.6 Gbps at a frequency of 168 MHz, with an area usage of 4039 slices.

F.-X. Standaert, et. al. [28] in 2003 presented a hardware implementation for MISTY1 block cipher. With the use of a Xilinx VIRTEX1000BG560 - 4 FPGA device achieved a throughput of 10.1 Gbps with a clock ratio of 159 MHz and 6322 slices area usage. Their design targeted on unrolling and pipelining the rounds of the cipher which resulted in having an output every cycle after the initial latency of 208 clock cycles.

2.1.13 KHAZAD

KHAZAD is a block cipher designed by Paulo S. L. M. Barreto together with Vincent Rijmen, one of the designers of the Advanced Encryption Standard (Rijndael). KHAZAD is named after Khazad-dûm, the fictional dwarven realm in the writings of J. R. R. Tolkien. KHAZAD was presented at the first NESSIE workshop in 2000, and, after some small changes, was selected as a finalist in the project.

KHAZAD has an eight-round substitution–permutation network with a 64-bit block size and a 128-bit key.[3]

F.-X. Standaert, et. al. [29] in 2002 published a hardware implementation for KHAZAD block cipher-focused high throughput circuits. By unrolling the cipher rounds and pipelining them they achieve a throughput of 9.4 Gbps in a Xilinx VIRTEX1000BG560 - 4 FPGA device, with a clock frequency of 148 MHz and a 8800 slices area usage.

2.1.14 Camellia

Camellia is a symmetric key block cipher with a block size of 128 bits and key sizes of 128, 192, and 256 bits. It was jointly developed by Mitsubishi Electric and NTT of Japan. The cipher has been approved for use by the ISO/IEC, the European Union's NESSIE project, and the Japanese CRYPTREC project. The cipher has security levels and processing abilities comparable to the Advanced Encryption Standard.

The cipher was designed to be suitable for both software and hardware implementations, from low-cost smart cards to high-speed network systems. It is part of the Transport Layer Security (TLS) cryptographic protocol designed to provide communications security over a computer network such as the Internet.

The cipher was named for the flower Camellia japonica, which is known for being long-lived as well as because the cipher was developed in Japan.

Camellia is a Feistel cipher with either 18 rounds (when using 128-bit keys) or 24 rounds (when using 192- or 256-bit keys). Every six rounds, a logical transformation layer is applied: the so-called "FL-function" or its inverse. Camellia uses four 8×8-bit S-boxes with input and output affine transformations and logical operations. The cipher also uses input and output key whitening. The diffusion layer uses a linear transformation based on a matrix with a branch number of 5.[3]

Z.Čiča [30] presented a pipelined implementation of the Camellia encryption algorithm. Using a Virtex5 XC5FX70T FPGA device achieved a throughput of 32.15 Gbps for encryption and 32.2 Gbps for decryption, with a clock frequency of 251.2 MHz and 251.6 MHz and with resource usage of 9753 (21%) slices, 3369 (7%) FF and 9753 (21%) slices, 3352 (7%) FF respectively.

2.1.15 ARIA

ARIA is a block cipher designed in 2003 by a large group of South Korean researchers. In 2004, the Korean Agency for Technology and Standards selected it as a standard cryptographic technique.

The algorithm uses a substitution–permutation network structure based on AES. The interface is the same as AES: 128-bit block size with key size of 128, 192, or 256 bits. The number of rounds is 12, 14, or 16, depending on the key size. ARIA uses two 8×8-bit S-boxes and their inverses in alternate rounds, one of these is the Rijndael S-box.

The key schedule processes the key using a 3-round 256-bit Feistel cipher, with the binary expansion of $1/\pi$ as a source of "nothing up my sleeve numbers". [3]

S.-W. Lee, et. al. [31] in 2008 presented a four-stage sub-pipelined architecture for ARIA block cipher achieving a throughput of 43 Gbps in a 0.25 μm CMOS technology, reaching 338 MHz clock frequency and using 260354 gates.

2.2 Conclusions of the Comparative Analysis

Based on the information we gathered above some of the algorithms are not able to meet the low limit of the desired throughput of 10 Gbps, as a result, they have been eliminated as an option for our implementation. These algorithms are MARS, IDEA, TEA, CAST - 128, and KHAZAD.

DES and TDES even though appeared to have some publications with great results, the fact that are "old technology" and are more vulnerable to attacks doesn't render them as a suitable option.

Considering the limited number of hardware implementation publications for some of the algorithms, we excluded XTEA, MISTY1, Camellia, and ARIA even though they show promising results.

All the above leaves us with five suitable options: AES, Serpent, RC6, Blowfish, and Twofish. For our first choice, we selected AES. The algorithm has an enormous number of software and hardware implementation in various technologies, with impressive results, and is used in several applications to this day. Serpent, even though appears to be a great competitor to AES, seems to show a larger area usage in comparison to AES. So we decided to exclude it. For our second choice, we selected Blowfish. As we can see Blowfish can achieve high throughput with a low area utilization which makes it a great option. Finally, we decided to also exclude Twofish based on the fact that has a bigger area utilization in comparison to Blowfish, which is his ancestor. This leaves us with our third and final choice the RC6 block cipher.

Chapter 3

Encryption/Decryption Algorithms

In this chapter, we will analyze the encryption/decryption algorithms implemented in our system and how each works. In Figure 3.1, we present a flowchart of our design to give a general idea of the way it operates as we proceed with the description of the individual parts.

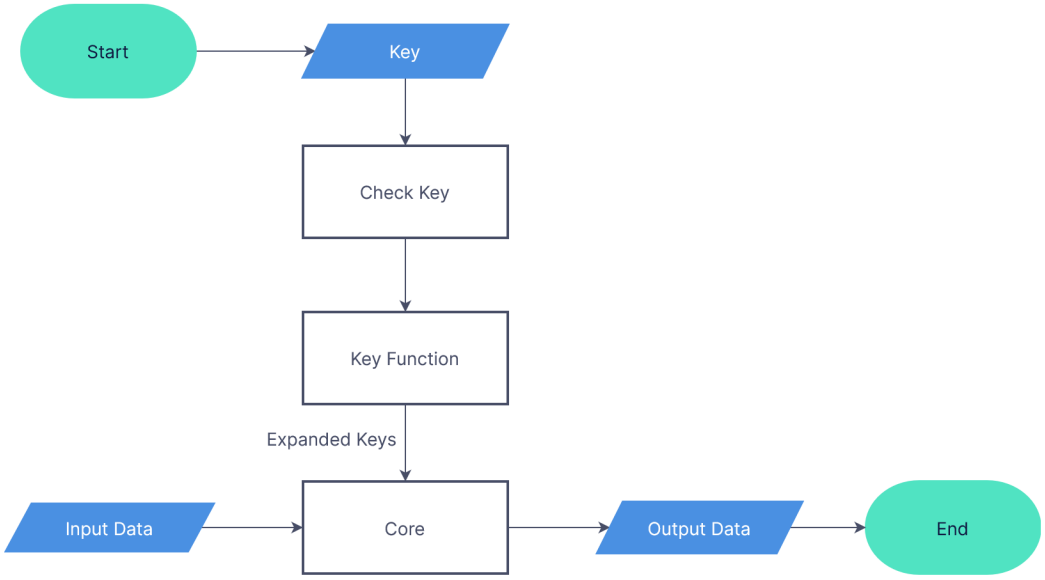


FIGURE 3.1: Flowchart of the Design

In particular, as we saw at the start, the key is used as input in the Check Key function (described in sec. 3.1). The result of the Check Key function is entered into the Key Function, along with the input key.

The Key Function is in charge of generating the Extended Keys that will be utilized later in the input data encryption or decryption process. Depending on the result of the Check Key function, the Key Function will either provide the Expended Keys that have been stored in case the key is the same as before

or will initiate the respective key expanding function of each algorithm if the key is new. Furthermore, we describe the individual key-expansion function of each algorithm - In sec. 3.2.1 for AES, in sec.3.3.1 for RC6, and in sec.3.4.1 for Blowfish.

The individual algorithm's (AES, RC6, or Blowfish) encryption or decryption procedure is carried out by the Core Function. The Input Data and Expanded Keys are entered into the Core, and the Output Data is created after the corresponding task is performed. Later on, we'll describe in detail the encryption and decryption processes of all algorithms - AES in sec.3.2.2 and sec.3.2.3, RC6 in sec.3.3.2, and Blowfish in sec.3.4.2 and sec.3.4.3.

3.1 Check Key

The Check Key functions works as a comparator. Its task is to identify if the incoming key is new or the same as the previous one and notify the Key Function with the result. This function is the same for every algorithm implemented.

3.2 Advanced Encryption Standard - AES

[32] For the AES algorithm, **the length of the input block, the output block, and the State is 128 bits**. This is represented by Nb = 4, which reflects the number of 32-bit words (number of columns) in the State.

For the AES algorithm, **the length of the Cipher Key, *K*, is 128, 192, or 256 bits**. The key length is represented by Nk = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key.

For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr, where Nr = 10 when Nk = 4, Nr = 12 when Nk = 6, and Nr = 14 when Nk = 8. The only Key-Block-Round combinations that conform to this standard are given in Table 3.1.

In our design, we consider the case of AES - 128, with Key Length 128 bits (Nk = 4), Block Size 128 bits (Nb = 4), and Number of Rounds Nr = 10.

For both its Cipher/Encryption and Inverse Cipher/Decryption, the AES algorithm uses a round function that is composed of four different byte-oriented transformations:

TABLE 3.1: Key-Block-Round Combinations.

	Key Length (Nk words)	**Block Size** (Nb words)	**Number of Rounds** (Nr)
AES - 128	4	4	10
AES - 192	6	4	12
AES - 256	8	4	14

1. byte substitution using a substitution table (S-box),
2. shifting rows of the State array by different offsets,
3. mixing the data within each column of the State array, and
4. adding a Round Key to the State.

3.2.1 AES Key Function

The AES algorithm takes the Cipher Key, **K**, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of $Nb(Nr+1)$ words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted $[w_i]$, with i in the range $0 \leq i < Nb(Nr+1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo-code in 1.

SubWord() is a function that takes a four-byte input word and applies the S-box (Sec. 3.2.2.1) to each of the four bytes to produce an output word. The function **RotWord()** takes a word $[a0, a1, a2, a3]$ as input, performs a cyclic permutation, and returns the word $[a1, a2, a3, a0]$. The round constant word array, **Rcon[i]**, contains the values given by $[x^{i-1}, \{00\}, \{00\}, \{00\}]$, with x^{i-1} being powers of x (x is denoted as $\{02\}$) in the field $GF(2^8)$, as discussed in Sec. A.3.2 (note that i starts at 1, not 0).

From the algorithm 1, it can be seen that the first Nk words of the expanded key are filled with the Cipher Key. Every following word, **w[i]**, is equal to the XOR of the previous word, **w[i-1]**, and the word Nk positions earlier, **w[i-Nk]**. For words in positions that are a multiple of Nk, a transformation is applied to **w[i-1]** prior to the XOR, followed by an XOR with a round constant, **Rcon[i]**. This transformation consists of a cyclic shift of the bytes in a

word (**RotWord()**), followed by the application of a table lookup to all four bytes of the word (**SubWord()**).

It is important to note that the Key Expansion routine for 256-bit Cipher Keys ($Nk = 8$) is slightly different than for 128- and 192-bit Cipher Keys. If $Nk = 8$ and $i - 4$ is a multiple of Nk, then **SubWord()** is applied to w[i-1] prior to the XOR.

Note that Nk=4, 6, and 8 do not all have to be implemented; they are all included in the conditional statement below for conciseness.

Algorithm 1 Pseudo Code for AES Key Expansion source:[32]

```
Nr : Number of rounds, which is a function of Nk and Nb (which is
fixed). For this standard, Nr = 10, 12, or 14.
Nk : Number of 32-bit words comprising the Cipher. Key For this
standard, Nk = 4, 6, or 8.
Nb : Number of columns (32-bit words) comprising the State. For
this standard, Nb = 4.

KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp

    i = 0

    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while

    i = Nk

    while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

3.2.2 AES Cipher / Encryption Core

At the start of the Cipher, the input is copied to the State array using the conventions described in Sec.A.2.4. After an initial Round Key addition, the State array is transformed by implementing a round function 10 times, with the final round differing slightly from the first 9 (Nr -1) rounds. The final State is then copied to the output as described in Sec. A.2.4.

The Cipher is described in the pseudo-code in 2. The individual transformations -**SubBytes()**, **ShiftRows()**, **MixColumns()**, and **AddRoundKey()** – process the State and are described in the following subsections.

Algorithm 2 Pseudo Code for the AES Cipher / Encryption source:[32]

```
Nr : Number of rounds, which is a function of Nk and Nb (which is
fixed). For this standard, Nr = 10, 12, or 14.
Nb : Number of columns (32-bit words) comprising the State. For
this standard, Nb = 4.

Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]

    state = in

    AddRoundKey(state, w[0, Nb-1])
    for round = 1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for

    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    out = state
end
```

3.2.2.1 SubBytes() Transformation

The SubBytes() transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box (Table 3.2), which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field GF(2^8), described in Sec.A.3.2; the element $\{00\}$ is mapped to itself.

2. Apply the following affine transformation (over GF(2)):

$$b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i \quad (3.1)$$

for $0 \leq i < 8$, where $\mathbf{b_i}$ is the **i^{th} bit of the byte**, and $\mathbf{c_i}$ is the **i^{th} bit of a byte c with the value** $\{63\}$ **or** $\{01100011\}$. Here and elsewhere, a prime on a variable (e.g., b') indicates that the variable is to be updated with the value on the right.

In matrix form, the affine transformation element of the S-box can be expressed as:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad (3.2)$$

Figure 3.2 illustrates the effect of the SubBytes()transformation on the State.

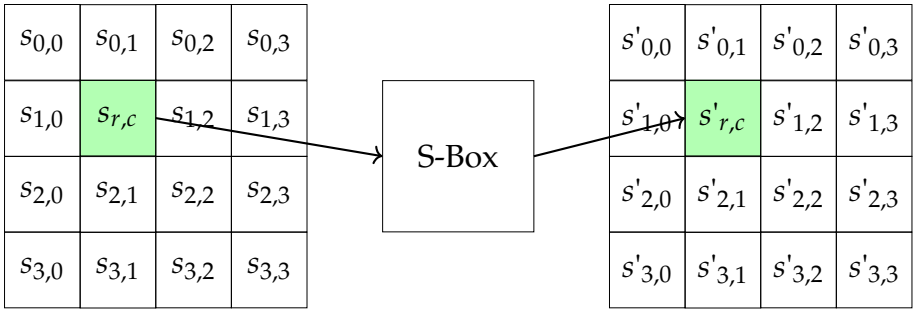


FIGURE 3.2: SubBytes() applies the S-box to each byte of the State. source : [32]

The S-box used in the **SubBytes()** transformation is presented in hexadecimal form in Table 3.2 . For example, if $s_{1,1} = \{53\}$, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Table 3.2 . This would result in $s'_{1,1}$ having a value of $\{ed\}$.

		y														
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
x	**0**	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

TABLE 3.2: S-box: substitution values for the byte xy (in hexadecimal format). source: [32]

3.2.2.2 ShiftRows() Transformation

In the **ShiftRows()** transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted.

Specifically, the **ShiftRows()** transformation proceeds as follows:

$$s'_{r,c} = s_{r,(c+shift(r,Nb)) \; mob \; Nb} \quad for \quad 0 < r < 4 \quad and \quad 0 \leq c < Nb, \tag{3.3}$$

where the shift value shift(r,Nb) depends on the row number, r, as follows (recall that Nb = 4):

$$shift(1,4) = 1; \quad shift(2,4) = 2; \quad shift(3,4) = 3. \tag{3.4}$$

This has the effect of moving bytes to "lower" positions in the row (i.e., lower values of c in a given row), while the "lowest" bytes wrap around into the "top" of the row (i.e., higher values of c in a given row).

Figure 3.3 illustrates the ShiftRows() transformation.

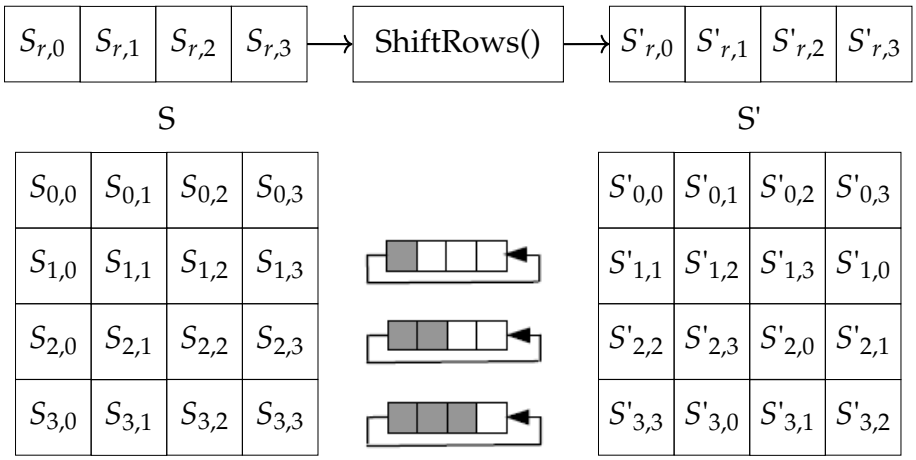


FIGURE 3.3: ShiftRows() cyclically shifts the last three rows in the State. source: [32]

3.2.2.3 MixColumns() Transformation

The **MixColumns()** transformation operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. A.3.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo x^4+1 with a fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^3 + \{01\}x^3 + \{02\}. \tag{3.5}$$

As described in Sec.A.3.3, this can be written as a matrix multiplication. Let $s'(x) = a(x) \oplus s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad for\, 0 \leq c < Nb. \tag{3.6}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}) \end{aligned} \tag{3.7}$$

Figure 3.4 illustrates the MixColumns()transformation.

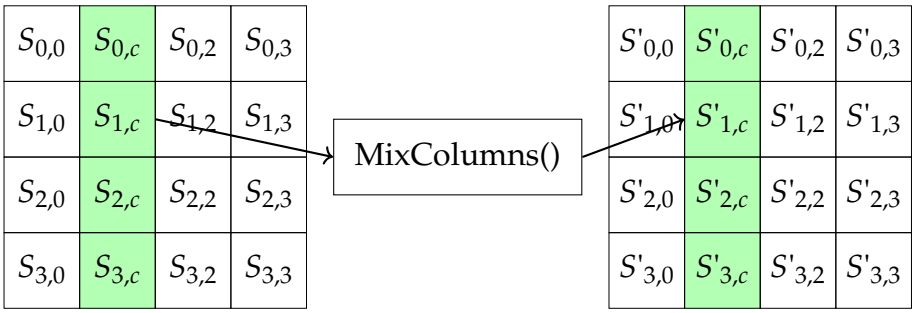


FIGURE 3.4: MixColumns() operates on the State column-by-column. source: [32]

3.2.2.4 AddRoundKey() Transformation

In the **AddRoundKey()** transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of Nb words from the key schedule (described in Sec....). Those Nb words are each added into the columns of the State, such that

$$[\,S'_{0,c}\,,\,S'_{1,c}\,,\,S'_{2,c}\,,\,S'_{3,c}\,] = [\,S_{0,c}\,,\,S_{1,c}\,,\,S_{2,c}\,,\,S_{3,c}\,] \oplus [\,W_{round*Nb+c}\,] \quad for\, 0 \leq c < Nb, \tag{3.8}$$

where $[w_i]$ are the key schedule words described in Sec...., and round is a value in the range $0 \leq round \leq Nr$. In the Cipher, the initial Round Key addition occurs when round = 0, prior to the first application of the round function (see alg. 2). The application of the **AddRoundKey()** transformation to the Nr rounds of the Cipher occurs when $1 \leq round \leq Nr$.

The action of this transformation is illustrated in Fig. 3.5, where $l = round * Nb$. The byte address within words of the key schedule was described in Sec. A.2.2.

3.2.3 AES Inverse Cipher / Decryption Core

The Cipher transformations in Sec.3.2.2 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - **InvShiftRows()**, **InvSubBytes()**, **InvMixColumns()**, and **AddRoundKey()** – process the State and are described in the following subsections.

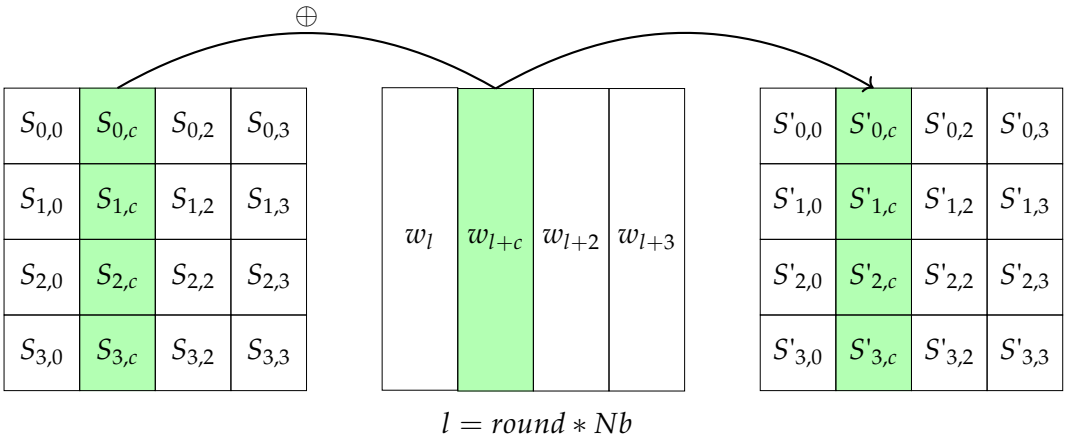


FIGURE 3.5: AddRoundKey() XORs each column of the State with a word from the key schedule. source: [32]

The Inverse Cipher is described in the pseudo-code in 3. In algorithm 3, the array $w[]$ contains the key schedule, which was described previously in Sec. 3.2.1.

3.2.3.1 InvShiftRows() Transformation

Specifically, the **InvShiftRows()** transformation proceeds as follows:

$$s_{r,(c+shift(r,Nb))\ mob\ Nb} = s'_{r,c} \quad for \quad 0 < r < 4 \quad and \quad 0 \leq c < Nb, \tag{3.9}$$

Figure 3.6 illustrates the **InvShiftRows()** transformation.

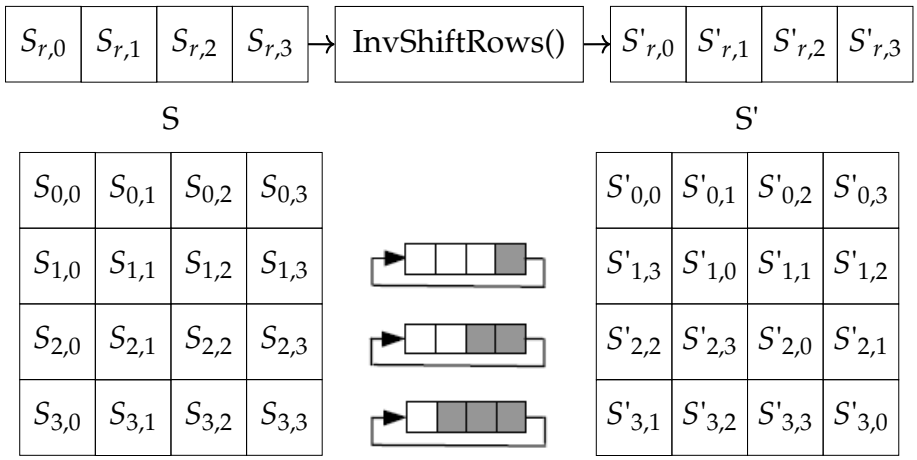


FIGURE 3.6: InvShiftRows() cyclically shifts the last three rows in the State. source: [32]

Algorithm 3 Pseudo Code for the AES Inverse Cipher / Decryption source:[32]

```
Nr : Number of rounds, which is a function of Nk and Nb (which is
fixed). For this standard, Nr = 10, 12, or 14.
Nb : Number of columns (32-bit words) comprising the State. For
this standard, Nb = 4.

InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]

    state = in

    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
    for round = Nr-1 step -1 downto 1
        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
        InvMixColumns(state)
    end for

    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[0, Nb-1])

    out = state
end
```

3.2.3.2 InvSubBytes() Transformation

InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (3.2.2) followed by taking the multiplicative inverse in GF(2^8). The inverse S-box used in the InvSubBytes() transformation is presented in Table 3.3:

		y														
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
x	**0**	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

TABLE 3.3: Inverse S-box: substitution values for the byte xy (in hexadecimal format). source: [32]

3.2.3.3 InvMixColumns () Transformation

InvMixColumns() is the inverse of the MixColumns() transformation. InvMixColumns() operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. A.3.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^3 + \{09\}x^3 + \{0e\}. \tag{3.10}$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let $s'(x) = a^{-1}(x) \oplus s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad for\, 0 \leq c < Nb. \tag{3.11}$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{0,c}) \\ s'_{1,c} &= (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{0,c}) \\ s'_{2,c} &= (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{3,c}) \end{aligned} \tag{3.12}$$

3.2.3.4 Inverse of the AddRoundKey() Transformation

AddRoundKey(), which was described in Sec. 3.2.2.4 is its own inverse since it only involves an application of the XOR operation.

3.3 RC6 - Rivest Cipher 6

[33] RC6 is a fully parameterized family of encryption algorithms. A version of RC6 is more accurately specified as RC6 -$w/r/b$ where the *word size* is w bits, encryption consists of a *non-negative number of rounds* r, and b denotes the *length of the encryption key in bytes*.

Since our chosen version of AES consists of 128-bit Block Size with a Key Length of 128-bits, the equivalent version of RC6 -$w/r/b$ is RC6 - *32/20/16*.

For all variants, **RC6 - w/r/b** operates on units of **four w-bit words** using the following six basic operations. The base-two logarithm of w will be denoted by $\lg w$

$a + b$ integer addition modulo 2^w

$a - b$ integer subtraction modulo 2^w

$a \oplus b$ bitwise exclusive-or of w-bit words

$a \times b$ integer multiplication modulo 2^w

$a \lll b$ rotate the w -bit word a to the left by the amount given by the least significant $\lg w$ bits of b

$a \ggg b$ rotate the w -bit word a to the right by the amount given by the least significant $\lg w$ bits of b

3.3.1 RC6 Key Expansion

The key schedule of RC6-$w/r/b$ is practically identical to the key schedule of RC5-$w/r/b$, the only difference is that more words are derived from the user-supplied key for use during encryption and decryption. The user supplies a key of b bytes. Sufficient zero bytes are appended to give a key length equal to a non-zero integral number of words; these key bytes are then loaded in little-endian fashion into an array of c w-bit words $L[0], ..., L[c-1]$. Thus the first byte of the key is stored as the low-order byte of $L[0]$, etc., and $L[c-1]$ is padded with high-order zero if necessary. (Note that if $b = 0$ then $c = 1$ and $L[0] = 0$). The number of w - bit words that will be generated for the additive round keys is $2r + 4$ and these are stored in the array $S[0, ..., 2r + 3]$.

The constants $P_{32} = B7E15163$ and $Q_{32} = 9E3779B9$ (hexadecimal) are the same " magic constants" as used in the RC5 key schedule. The value of P_{32} is derived from the binary expansion of $e - 2$, where e is the base of the natural

logarithm function. The value of Q_{32} is derived from the binary expansion of $\phi - 1$, where ϕ is the Golden Ratio. Similar definitions from RC5 for P_{64} etc. can be used for versions of RC6 with other word sizes. These values are somewhat arbitrary, and other values could be chosen to give "custom" or proprietary versions of RC6.

Algorithm 4 Pseudo Code For Key Schedule for RC6 - $w/r/b$. source:[33]

Key schedule for RC6 – $w/r/b$
where w : word size in bits
r : number of rounds
b : length of the encryption key in bytes

Input: User–supplied b byte key preloaded into the c–word array $L[0, ..., c-1]$
Number r of rounds

Output: w – bit round keys $S[0, ..., 2r+3]$

Procedure: $S[0] = P_w$

for $i = 1$ to $2r+3$ do
$S[i] = S[i-1] + Q_w$

$A = B = i = j = 0$

$v = 3 \times max\{c, 2r+4\}$
for $s = 1$ to v do
{
$A = S[i] = (S[i] + A + B) \lll 3$
$B = L[j] = (L[j] + A + B) \lll (A + B)$
$i = (i+1) \mod (2r+4)$
$j = (j+1) \mod c$

}

3.3.2 RC6 Encryption and Decryption Core

RC6 works with four w -bit registers A, B, C, D which contain the initial input plaintext as well as the output ciphertext at the end of encryption. The first byte of plaintext or ciphertext is placed in the least-significant byte of A; the last byte of plaintext or ciphertext is placed into the most-significant byte of D. We use $(A, B, C, D) = (B, C, D, A)$ to mean the parallel assignment of values on the right to registers on the left.

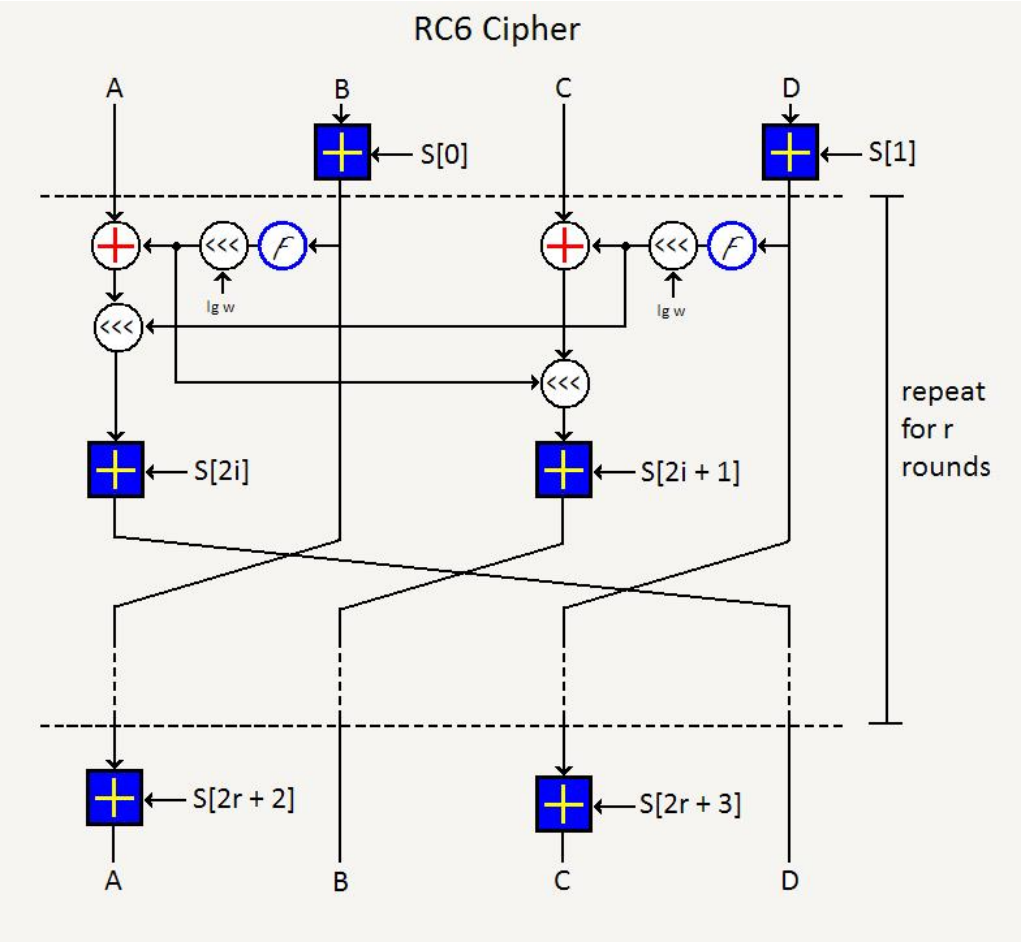


FIGURE 3.7: Encryption with RC6 - w /r /b. Here $f(x) = x \times (2x + 1)$.

source : https://en.wikipedia.org/wiki/RC6

Algorithm 5 Pseudo Code For Encryption with RC6 - $w/r/b$ source:[33]

Encryption with RC6 – $w/r/b$
where w : word size in bits
r : number of rounds
b : length of the encryption key in bytes

Input: Plaintext stored in four w – bit input registers A, B, C, D
Number r of rounds
w – bit round keys $S[0, ..., 2r+3]$

Output: Ciphertext stored in A, B, C, D

Procedure:
$B = B + S[0]$
$D = D + S[1]$
for i = 1 to r do
{
$t = (B \times (2B+1)) \lll \lg w$
$u = (D \times (2D+1)) \lll \lg w$
$A = ((A \oplus t) \lll u) + S[2i]$
$C = ((C \oplus u) \lll t) + S[2i+1]$
$(A, B, C, D) = (B, C, D, A)$
}
$A = A + S[2r+2]$
$C = C + S[2r+3]$

Algorithm 6 Pseudo Code For Decryption with RC6 - $w/r/b$ source:[33]

Decryption with RC6 – $w/r/b$
where w : word size in bits
r : number of rounds
b : length of the encryption key in bytes

Input: Ciphertext stored in four w –bit input registers A, B, C, D
Number r of rounds
w – bit round keys $S[0, ..., 2r+3]$

Output: Plaintext stored in A, B, C, D

Procedure:
$C = C + S[2r+3]$
$A = A + S[2r+2]$
for i = 1 to r do
{
$(A, B, C, D) = (D, A, B, C)$
$u = (D \times (2D+1)) \lll \lg w$
$t = (B \times (2B+1)) \lll \lg w$
$C = ((C - S[2i+1]) \ggg t) \oplus u$
$A = ((A - S[2i]) \ggg u) \oplus t$
}
$D = D + S[1]$
$B = B + S[0]$

3.4 Blowfish

[34] Blowfish is a variable-length key, 64-bit block cipher. The algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key of 32 bits up to 448 bits into several subkey arrays totaling 4168 bytes.

In our design, the version of Blowfish being implemented consists of a 128-bit key size.

Data encryption occurs via a 16-round Feistel network. Each round consists of a key-dependent permutation, and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round.

Subkeys:

Blowfish uses a large number of subkeys. These keys must be precomputed before any data encryption or decryption.

1. The P-array The P-array consists of 18 32-bit subkeys:
   $P_1, P_2, ..., P_{18}$
2. There are four 32-bit S-boxes with 256 entries each:
   $S_{1,0}, S_{1,1}, ..., S_{1,255}$
   $S_{2,0}, S_{2,1}, ..., S_{2,255}$
   $S_{3,0}, S_{3,1}, ..., S_{3,255}$
   $S_{4,0}, S_{4,1}, ..., S_{4,255}$

The exact method used to calculate these subkeys is described in Sec 3.4.1.

3.4.1 Blowfish Key Function

The subkeys are calculated using the Blowfish algorithm (described at Sec. 3.4.2). The exact method is as follows:

1. Initialize first the P-array and then the four S-boxes, in order, with a fixed string. This string consists of the hexadecimal digits of pi (less the initial 3). For example:
   $P_1 = 0x243f6a88$
   $P_2 = 0x85a308d3$
   $P_3 = 0x13198a2e$

$P_4 = 0x03707344$

2. XOR P_1 with the first 32 bits of the key, XOR P_2 with the second 32-bits of the key, and so on for all bits of the key (possibly up to P_{14}). Repeatedly cycle through the key bits until the entire P-array has been XORed with key bits. (For every short key, there is at least one equivalent longer key; for example, if A is a 64-bit key, then AA, AAA, etc., are equivalent keys.)
3. Encrypt the all-zero string with the Blowfish algorithm, using the subkeys described in steps (1) and (2).
4. Replace P_1 and P_2 with the output of step (3).
5. Encrypt the output of step (3) using the Blowfish algorithm with the modified subkeys.
6. Replace P_3 and P_4 with the output of step (5).
7. Continue the process, replacing all entries of the P- array, and then all four S-boxes in order, with the output of the continuously-changing Blowfish algorithm.

In total, 521 iterations are required to generate all required subkeys. Applications can store the subkeys rather than execute this derivation process multiple times.

3.4.2 Blowfish Encryption Core

Blowfish is a Feistel network consisting of 16 rounds (see Figure 3.8). The input is a 64-bit data element, x.

Algorithm 7 Pseudo Code For Blowfish Encryption source:[34]

Divide x into two 32-bit halves: x_L, x_R
For $i = 1$ step 1 to 16:
    $x_L = x_L \oplus P_i$
    $x_R = F(x_L) \oplus x_R$
    Swap x_L and x_R
Swap x_L and x_R (Undo the last swap.)
$x_R = x_R \oplus P_{17}$
$x_L = x_L \oplus P_{18}$
Recombine x_L and x_R

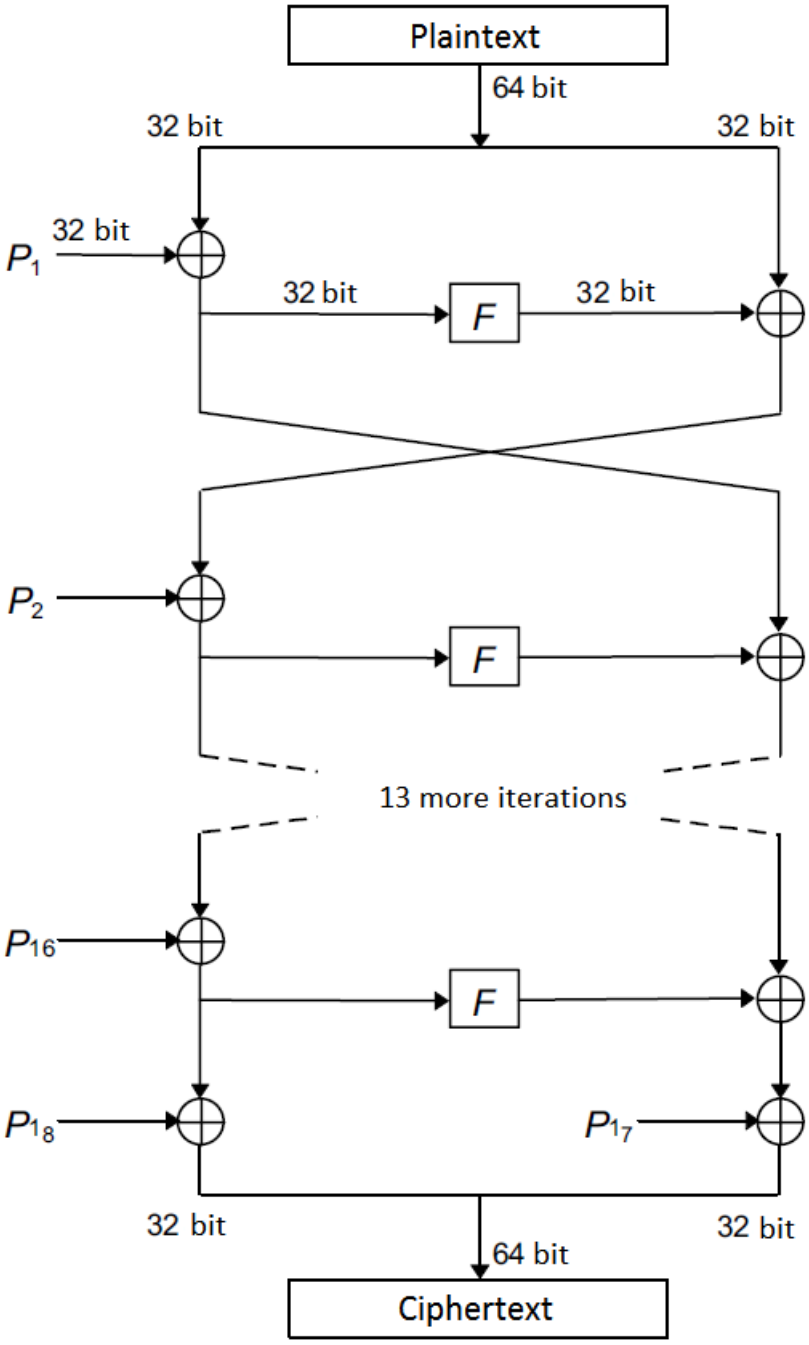


FIGURE 3.8: Blowfish Encryption Block Diagram. source:[34]

Function F (see Figure 3.9):

Divide x_L into four eight-bit quarters: a, b, c, and d
$F(x_L) = ((S_{1,a} + S_{2,b} \mod 2^{32}) \oplus S_{3,c}) + S_{4,d} \mod 2^{32}$

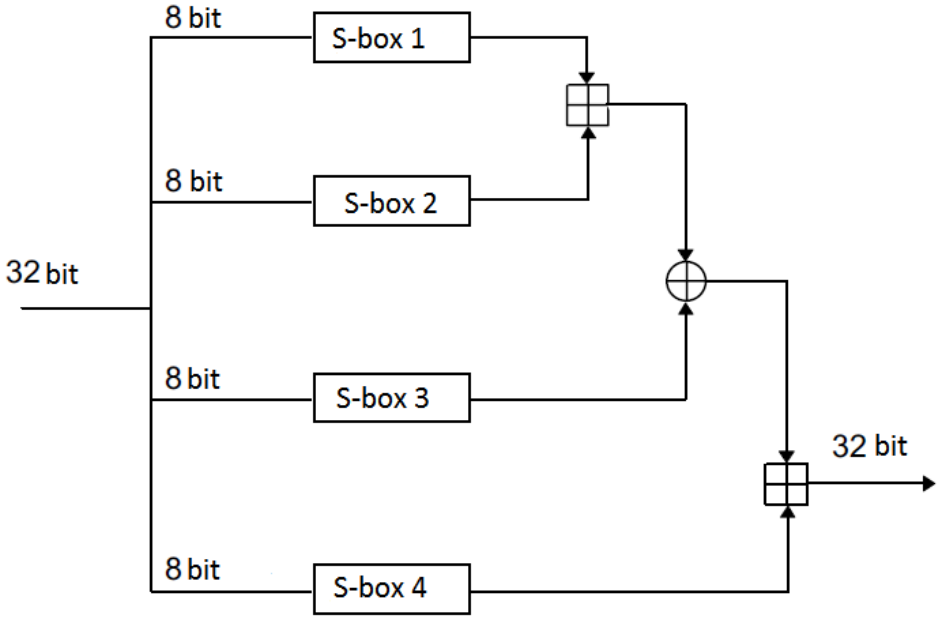


FIGURE 3.9: Function F. source:[34]

3.4.3 Blowfish Decryption Core

Decryption is exactly the same as encryption, except that P_1, P_2 ..... P_{18} are used in the reverse order (see Figure 3.10).

Algorithm 8 Pseudo Code For Blowfish Decryption source:[34]

Divide x into two 32-bit halves: x_L, x_R
For $i = 18$ step -1 downto 3:
    $x_L = x_L \oplus P_i$
    $x_R = F(x_L) \oplus x_R$
    Swap x_L and x_R
Swap x_L and x_R (Undo the last swap.)
$x_R = x_R \oplus P_2$
$x_L = x_L \oplus P_1$
Recombine x_L and x_R

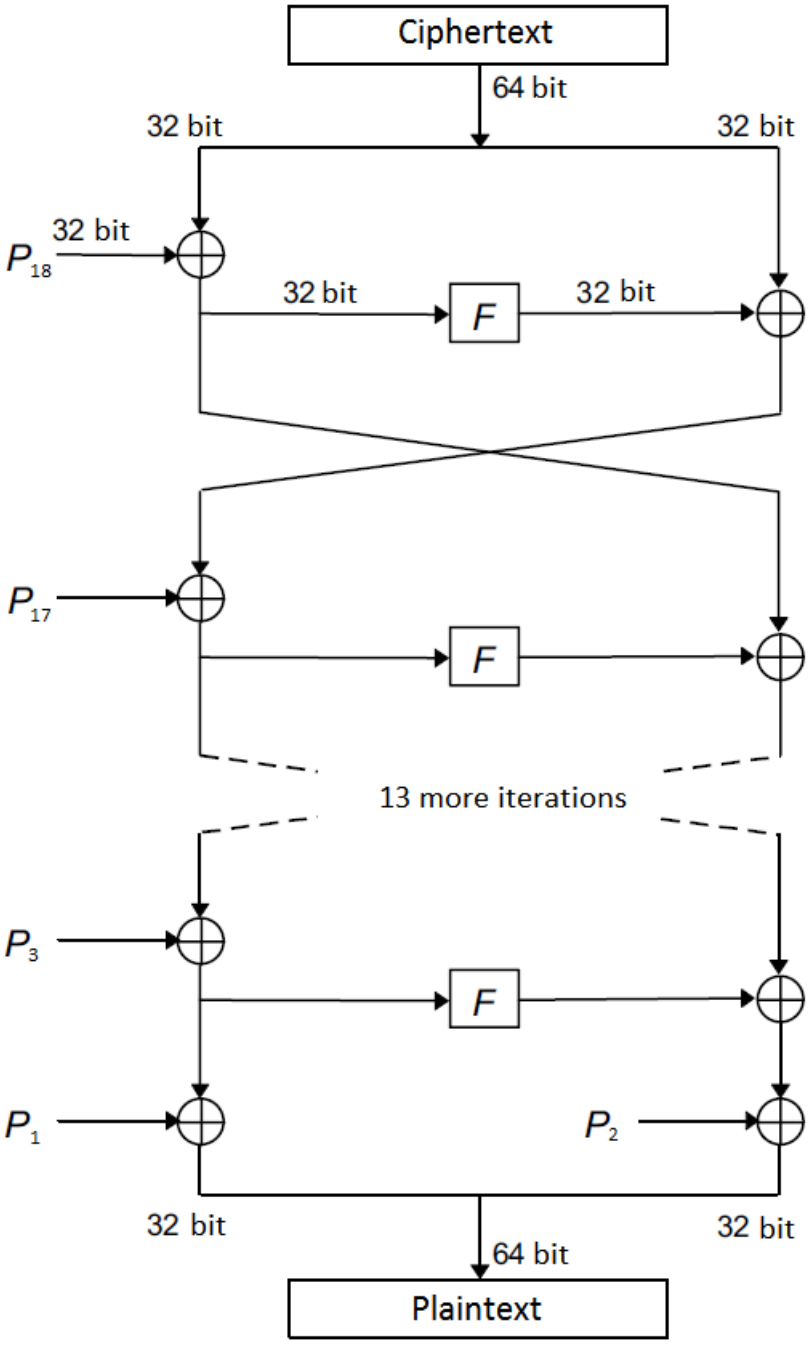


FIGURE 3.10: Blowfish Decryption Block Diagram

Chapter 4

Architecture and Detailed Design of AES Enryption

In this chapter, we will analyze the steps we took to implement the AES Encryption algorithm in the Vivado HLS environment (2017.1 version), as well as the optimizations we used to improve the overall performance (throughput, initial latency, and area utilization).

4.1 Vivado High-Level Synthesis (HLS)

The Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that you can synthesize into a Xilinx field programmable gate array (FPGA). You can write C specifications in C, C++, SystemC, or as an Open Computing Language (OpenCL™) API C kernel, and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power over traditional processors.[35]

High-level synthesis bridges hardware and software domains, providing the following primary benefits:

- Improved productivity for hardware designers

  Hardware designers can work at a higher level of abstraction while creating high-performance hardware.

- Improved system performance for software designers

  Software developers can accelerate the computationally intensive parts of their algorithms on a new compilation target, the FPGA.

Using a high-level synthesis design methodology allows us to:

- Develop algorithms at the C-level

  Work at a level that is abstract from the implementation details, which consumes development time.

- Verify at the C-level

  Validate the functional correctness of the design more quickly than with traditional hardware description languages.

- Control the C synthesis process through optimization directives

  Create specific high-performance hardware implementations.

- Create multiple implementations from the C source code using optimization directives

  Explore the design space, which increases the likelihood of finding an optimal implementation.

- Create readable and portable C source code

  Retarget the C source into different devices as well as incorporate the C source into new projects.

4.1.1 Synthesis Report

High-level synthesis creates the optimal implementation based on default behavior, constraints, and any optimization directives we specify. We can use optimization directives to modify and control the default behavior of the internal logic and I/O ports. This allows us to generate variations of the hardware implementation from the same C code.

To determine if the design meets our requirements, we can review the performance metrics in the synthesis report generated by high-level synthesis. After analyzing the report, we can use optimization directives to refine the implementation. The synthesis report contains information on the following performance metrics:

- **Area**: Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers, block RAMs, and DSP48s.
- **Latency**: Number of clock cycles required for the function to compute all output values.

- **Initiation interval (II)**: Number of clock cycles before the function can accept new input data.
- **Loop iteration latency**: Number of clock cycles it takes to complete one iteration of the loop.
- **Loop initiation interval**: Number of clock cycles before the next iteration of the loop starts to process data.
- **Loop latency**: Number of cycles to execute all iterations of the loop.

4.1.2 Directives

This section outlines various optimizations and techniques we can use to direct Vivado HLS to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource utilization of the resulting RTL code.

Some of these optimizations used in our project are described below.

- **pragma HLS inline**

  Removes a function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the register transfer level (RTL). In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations

- **pragma HLS interface**

  Specifies the interface protocol mode for function arguments, global variables used by the function, or the block-level control protocols.

- **pragma HLS pipeline**

  The PIPELINE pragma reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations.

- **pragma HLS unroll**

  Unroll loops to create multiple independent operations rather than a single collection of operations. The UNROLL pragma transforms loops by creating multiple copies of the loop body in the register transfer level (RTL) design, which allows some or all loop iterations to occur in parallel.

- **pragma HLS allocation**

  Specifies instance restrictions to limit resource allocation in the implemented kernel. This defines and can limit the number of register transfer level (RTL) instances and hardware resources used to implement specific functions, loops, operations, or cores.

- **pragma HLS array_map**

  Combines multiple smaller arrays into a single large array to help reduce block RAM resources.

- **pragma HLS array_partition**

  Partitions an array into smaller arrays or individual elements and provides the following: Results in RTL with multiple small memories or multiple registers instead of one large memory. Effectively increases the amount of read and write ports for storage. Potentially improves the throughput of the design. Requires more memory instances or registers.

- **pragma HLS dataflow**

  The DATAFLOW pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the register transfer level (RTL) implementation, and increasing the overall throughput of the design. Used to minimize interval.

- **pragma HLS stream**

  Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization.

4.2 Optimizations

Based on the AES Encryption as an example we will describe step by step the stages we took to optimize the overall performance of the algorithm.

Initially, we focused on the main body of the algorithm and decided to keep the key steady at the same value for the time. The functions for the creation of the expanded keys (KeyExpansion, KeyExpansionCore, and SubWord) are not being used, and the expanded keys needed for the encryption process are stored in an array. The project also includes a number of arrays needed for

the calculation of the SubBytes and the MixColumns transformations, as well as an array for the Rcon values needed for the key functions, for later use.

The initial version uses the following directives in its functions:

AES_Full_axis (Top Function)
At first, the top function uses the directive ***#pragma HLS inline region***. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function, but not inlining recursively through the hierarchy.

The ***#pragma HLS interface*** directive is being used to determine the I/O (input-output) protocol for the input and output arguments. For the arguments stream_in and stream_out of our top function, the mode is set to ***axis*** which implements those ports as an AXI4-Stream interface.

We used the directive ***#pragma HLS pipeline*** set to ***II = 1***. This means that the function processes new inputs every clock cycle. Vivado HLS tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval.

AES_Encrypt

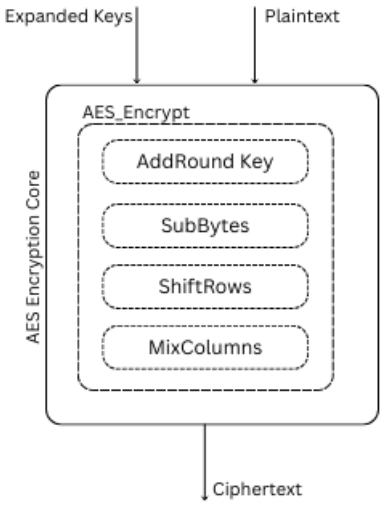


FIGURE 4.1: AES Encryption Core Block Diagram

The AES_Encrypt is responsible for the proper use of the individual functions needed to perform the encryption process which is described in sec.3.2.2.

Firstly, we use the ***#pragma HLS inline region*** directive. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function, but not inlining recursively through the hierarchy.

To determine the I/O (input-output) protocol for their input and output arguments, we use the ***#pragma HLS interface*** directive. For the plaintext and ciphertext, the mode is set to ***axis*** which implements those ports as an AXI4-Stream interface, as well.

Also, we include the directive ***#pragma HLS pipeline*** set to ***II = 1***. This means that they process new inputs every 1 clock cycle.

In the function, we also use the ***#pragma HLS allocation*** directive for their individual functions to determine the number of RTL instances and the hardware resources used to implement those. More specifically, we set the ***limit*** to 11 for the AddRoundKey, 10 for SubBytes, and 9 for MixColumns.

Additionally, we included the directive ***#pragma HLS array_map*** for the s_box and the arrays that keep the values needed for the SubBytes and the MixColumns operations, respectively. This directive combines those arrays into the same target ***instance = cipher*** in ***horizontal*** mapping. This corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.

Finally, we use the ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculation to be implemented in parallel.

AddRoundKey
The function AddRoundKey initially uses the directive ***#pragma HLS inline off*** so as to NOT be inlined upward into the calling functions or regions.

It also includes the directive ***#pragma HLS unroll*** to store the results in the state array in parallel.

MixColumns
The MixColumns function uses the directive ***#pragma HLS inline off***. This means that the functions should NOT be inlined upward into any calling functions or regions.

It also uses the directive ***#pragma HLS unroll*** for storing its results in the state array in parallel.

SubBytes

The SubBytes function uses the directive ***#pragma HLS inline off***. This means that the function should NOT be inlined upward into any calling functions or regions.

It also uses the directive ***#pragma HLS unroll*** for storing their results in the state array in parallel.

ShiftRows

In this function, the directive ***#pragma HLS unroll*** is being used for parallel storing the results in the state array.

Note: All the data types of the variable are defined as ap_uint<N>, where N is the number of bits. The size is determined by the nature of each variable.

The Synthesis Report after those changes is shown in fig.4.2. As can be seen, the latency is reduced and Interval = 1 is achieved. We also see an increase in the use of BRAMs as the number of instances required to attain interval = 1 grows.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	1.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
32	32	1	1	function

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	16	-
FIFO	-	-	-	-	-
Instance	224	-	4028	5396	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	66	-
Register	-	-	583	-	-
Total	224	0	4611	5478	0
Available	1824	2520	548160	274080	0
Utilization (%)	12	0	~0	1	0

FIGURE 4.2: Initial Synthesis Report of AES Encryption

4.2.1 Optimization Phase One

In this phase, our primary goal was to reduce the number of BRAMs being used. After analyzing the code, we discover that the major part of BRAMs is being used by the MixColumns function.

After deleting the arrays that kept the values needed in the MixColumns calculations and we created two new functions to produce the same results using multiplications with Galois Field A.3.3.

The Synthesis Report after those changes is shown in fig.4.3. As can be seen, we achieved a decrease in the usage of BRAMs, and of Flip Flops, but we had an increase in the use of LUTs due to the fact that Galois Field multiplications

consist of a series of XOR operations. Additionally, we had a reduction in the latency.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	1.48	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
23	23	1	1	function

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	16	-
FIFO	-	-	-	-	-
Instance	80	-	2858	8222	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	66	-
Register	-	-	565	-	-
Total	80	0	3423	8304	0
Available	1824	2520	548160	274080	0
Utilization (%)	4	0	~0	3	0

FIGURE 4.3: First Step Synthesis Report of AES Encryption

4.2.2 Optimization Phase Two

In this stage, we initiated the use of the key functions. We created a top function for the keys called KeyFunction, that handles the other ones needed for this process (KeyExpansion, KeyExpansionCore, and SubWord). We removed the array that kept the old expanded keys and replaced it with eleven static variables to store the new expanded keys after the key functions produce them.

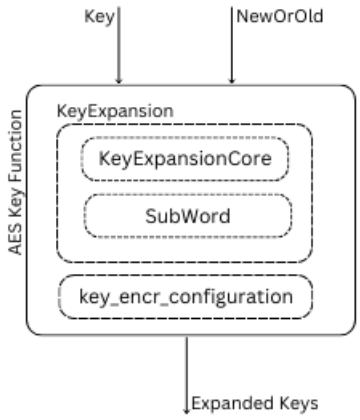


FIGURE 4.4: AES Key Function

For this task, we created the key_encr_configuration function, whose job is to take the results of the KeyExpansion and stored them in the static variables. This function also resolves a conflict between read and write the expanded keys at the same clock cycle.

The directives used in the KeyFunction are the following. The directive ***#pragma HLS inline off*** is being used so as to not be inlined upward into the calling function. We also add the directive ***#pragma HLS pipeline*** set to ***II = 1*** as to keep the desired target of the pipeline to 1 clock cycle. Additionally, we use the ***#pragma HLS array_partition*** for the ***variable = expandedKey*** by ***complete*** to partition it completely into individual elements. The expandedKey is the array holding the results of the KeyExpansion function. Finally, the use of the ***#pragma HLS unroll*** is needed for the for-loop used for the separation of the expandedKey array into the eleven variables.

Further, we perform some changes in the AddRoundKey function. At this stage, AddRoundKey no longer produces its results using the arrays storing the expanded keys, but the static variables created for this purpose. As a result, the ***#pragma HLS array_map*** directive is no longer needed, and it's being removed.

At this point, we change the variable type of the Top Function AES_Full_axis to stream <axiWord>, where axiWord is a struct containing the variables ap_uint<128> data, ap_uint<16> strb, and ap_uint<1> last, needed for the proper use of the AXI4 Stream Protocol.

Finally, we added two nested FSMs in the top function. The first one is for the separation of the key expansion process from the data encryption process. The other one is to continuously keep processing the data through encryption if there is valid data available.

Also, we included a register to keep the value of the key, so we can check if the incoming key is new or the same as before, and determine the need to call the Key Function or not.

The Synthesis Report after this step is shown in fig.4.5. We notice that unfortunately we lost the desired pipeline of one cycle, and the rest of the synthesis results don't represent the real numbers since the project doesn't work in the way we want. At this point, a major issue appeared, a warning said that the input/output variables of our top function contain leftover data, which may result in RTL simulation hanging.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	3.77	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
13	13	11	11	function

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	121	-
FIFO	-	-	-	-	-
Instance	38	-	7925	7576	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	1752	-
Register	-	-	1974	-	-
Total	38	0	9899	9449	0
Available	1824	2520	548160	274080	0
Utilization (%)	2	0	1	3	0

FIGURE 4.5: Second Step Synthesis Report of AES Encryption

4.2.3 Optimization Final Phase

At the final stage of the project's optimizations, we need to address the issue of the leftover data and the pipeline. This made us reconsider the overall design. The final design is shown in fig.4.6.

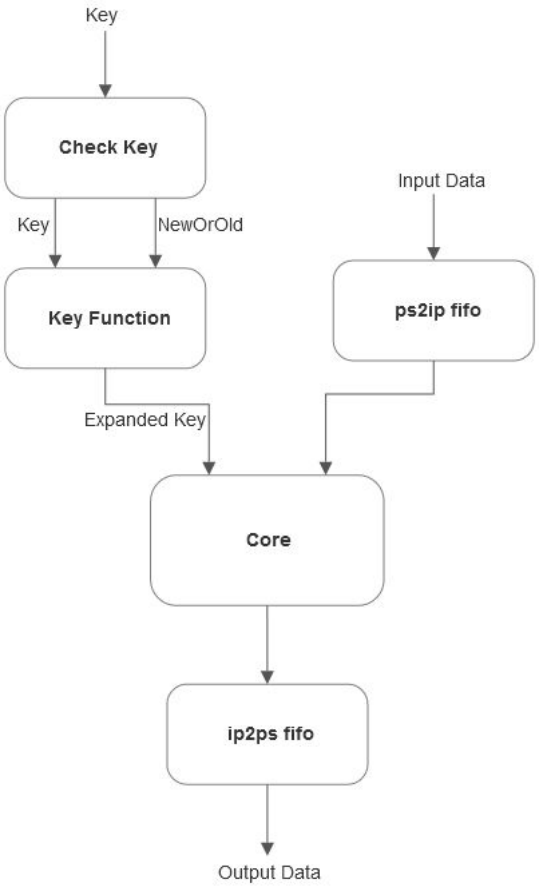


FIGURE 4.6: Top Module Block Diagram

my_ip_hls function (Top Function)

The top-level function of our project is my_ip_hls. It takes the variables key and slaveIn as inputs and masterOut as outputs (all variables are stream axiWord> types). The directive ***#pragma HLS DATAFLOW*** is included in the function. The ***DATAFLOW*** pragma enables task-level pipelining, which allows functions and loops to overlap in their operation, enhancing RTL concurrency and overall design performance.

For all three of its input/output variables, the directive ***#INTERFACE axis register both port = <name>*** is being used. This signifies that the function's arguments are all implemented as AXI4-Stream interfaces. The directive ***#INTERFACE ap_ctrl_none port = return*** was also added to prevent the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_done) from being formed.

Due to the dataflow directive the use of the ***#pragma HLS STREAM variable=<variable> depth=<int> dim=<int>*** is needed. It was used to generate a FIFO for each of the function's three arguments. In particular, we have a 64-depth fifo dimension 1 for the input and output data, and a 4-depth fifo dimension 1 for the key.

Finally, in this function, the calls of the rest of our design functions are being performed, as well as the interconnections of their variables.

Check Key Function

The Check Key function was created to identify in an early stage if the incoming key is new or is the same as the previous one that is stored in the key register. It notifies the Key Function with the result. The need for this function is to avoid creating additional latency.

It consists of an FSM that continuously reads if there is a new valid incoming key. The directive ***#pragma HLS pipeline II = 1*** is being used, as to be able to process new data every one clock cycle. Additionally, the directive ***#pragma HLS inline off*** is set to prevent the function to be inlined upward into the calling function.

Key Function

The Key Function creates the expanded keys in the same way described in phase 2. We added two FSMs, the first one to continuous reading if we have new incoming valid keys due to the nature of the protocol our project is based on. The second one exists to determine, based on the result of the Check Key

function, if the expanded keys needed for the encryption process are the ones stored in the static variables or if we need to initiate the process to create new ones.

Core Function

The Core Function is the renamed version of our previous top function AES_-Full_axis, and it's no longer the top function. At this stage, the function takes as input variables the input data from the ps2ip FIFO as well as the expanded keys calculated from the Key Function, and after producing its results sends them to the ip2ps fifo.

The function contains one FSM, that continuously reads if we have new incoming valid data, and calls the AES_Encrypt function to perform the encryption process. In this function the directive ***#pragma HLS pipeline II=1 enable_flush*** is being used, as to be able to process new data every one cycle, and if the data valid at the input of the pipeline goes inactive to flush and empty the pipeline.

Additionally, we removed the ***#pragma HLS allocation*** directive for the three functions AddRoundKey, SubBytes, and MixColumns and let Vivado automatically decide the number of instances needed to be created. We also removed from the AddRoundKey function the ***#pragma HLS inline off*** directive, and let the AES_Encrypt absorb the function. As a result of this change, the number of flip-flops and LUTs was slightly reduced.

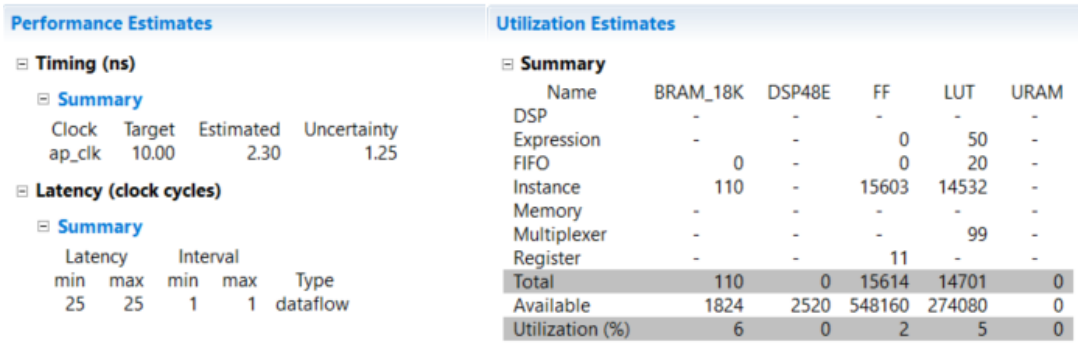
We also set the directive ***#pragma HLS INTERFACE ap_ctrl_none port = return***. Function-level protocols, also called block-level I/O protocols, provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The ***ap_ctrl_none*** means that the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_-done) are not created. Block-level I/O protocols can be assigned to a port for the function ***return*** value.

It also contains the ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculations be implemented in parallel.

ps2ip fifo and ip2ps fifo

The ps2ip and the ip2ps are 64-depth FIFOs that keep the input and output data respectively. The ps2ip FIFO takes the input data and promotes them into the Core function, and the ip2ps FIFO takes the results of the Core function and sends them as output data.

The Synthesis Report of our final design is shown in fig.4.7. As we can see, we successfully restored the Initiation Interval to 1, meaning that the project processes new data and has a new outcome every 1 clock cycle. Our final latency is 25, which means that our first result will be produced after 25 clock cycles. Furthermore, the overall usage of the resources is kept at a low level.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.30	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
25	25	1	1	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	0	-	0	20	-
Instance	110	-	15603	14532	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	99	-
Register	-	-	11	-	-
Total	110	0	15614	14701	0
Available	1824	2520	548160	274080	0
Utilization (%)	6	0	2	5	0

FIGURE 4.7: Final Step Synthesis Report of AES Encryption

4.3 Summary

In this chapter, as previously stated, we analyzed the phases of the development and optimization of the AES Encryption, starting from the beginning and ending with the final framework. In each phase, we applied a variety of directives that work best to optimize these types of algorithms, such as i) inline, ii) unroll, iii) pipeline, ..., etc. In table 4.1 we summarized all the directives we met throughout this process and displayed which of these are included in our final form.

Directives	***AES Encryption***
inline	✓
allocation	✗
unroll	✓
array_map	✗
array_partition	✓
interface	✓
pipeline	✓
dataflow	✓
stream	✓

TABLE 4.1: Directives of AES Encryption

Chapter 5

Hardware Architectures of AES Decryption, RC6 Encryption, RC6 Decryption, Blowfish Encryption, and Blowfish Decryption

For the purpose of this thesis, we developed six projects, two for each of the three algorithms we implemented: AES, RC6, and Blowfish, one for their encryption and one for their decryption procedures, respectively.

In the previous chapter, we analyzed the creation of the AES Encryption and how we ended up with its final framework. For the rest of the projects, a similar procedure was followed. Our main goal was to implement the rest of them based on the same framework described by the block diagram shown in fig.4.6.

In the following image, we demonstrate a more detailed version of the block diagram.

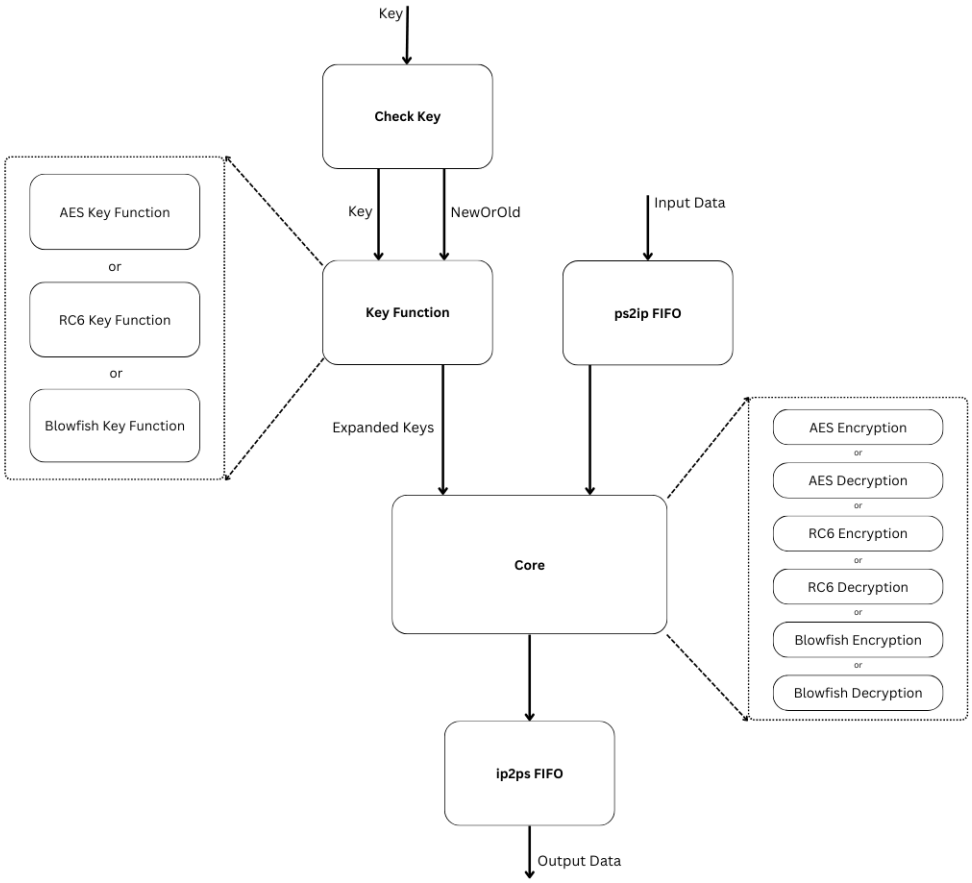


FIGURE 5.1: Detailed Version of the Block Diagram

In this chapter, we will analyze the architecture of the final form of each of these algorithms and which optimizations were used.

5.1 AES Decryption

The AES Decryption project, since it's the reverse of AES Encryption, has a similar structure and fits ideally in the targeted framework. In this section, we will analyze its functions and the directives being used. The block diagram of the design is demonstrated in fig. 5.1.

my_ip_hls function (Top Function)

The top-level function for the AES Decryption project is my_ip_hls. It takes the variables key and slaveInDec as inputs and masterOutDec as outputs. The directive ***# pragma HLS DATAFLOW*** is used in the function. The ***DATAFLOW*** pragma enables task-level pipelining, which allows functions and

loops to overlap in their operation, enhancing RTL concurrency and overall design performance.

Again in this project, for all three of its input/output variables, the directive ***#INTERFACE axis register both port = <name>*** is being used. To signify that the functions' arguments are all implemented as AXI4-Stream interfaces. The directive ***#INTERFACE ap_ctrl_none port = return*** was also added to prevent the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_done) from being formed.

Due to the dataflow directive the use of the ***#pragma HLS STREAM variable=<variable> depth=<int> dim=<int>*** is needed. It was used to generate a FIFO for each of the function's three arguments. In particular, we have a 64-depth FIFO dimension of 1 for the input and output data and a 4-depth FIFO dimension of 1 for the key.

Finally, in this function, the calls of the rest of our design functions are being performed, as well as the interconnections of their variables. Note that, all variables are type **stream <axiWord>**, where axiWord is a struct containing the variables ap_uint<128> data, ap_uint<16> strb, and ap_uint<1> last.

Check Key Function

The Check Key function is exactly the same as the one in the AES Encryption project, described in sec.4.2.3.

Key Function

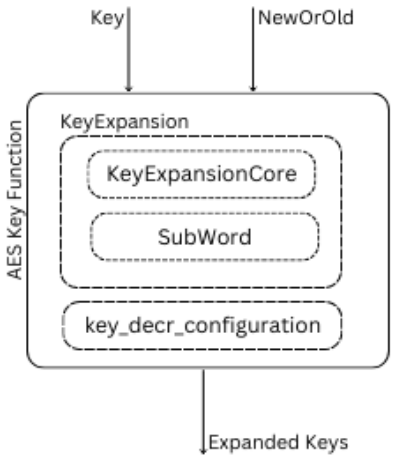


FIGURE 5.2: AES Decryption Key Function Block Diagram

The Key Function is the same as the one in AES Encryption. In detail, it consists of two FSMs, the first one to continuous reading if we have new incoming valid keys due to the nature of the protocol our project is based on. The second one exists to determine, based on the result of the Check Key function, if the expanded keys needed for the encryption process are the ones we stored or if we need to initiate the process to create new ones.

When there is the need to generate new expanded keys, the functions KeyExpansion, KeyExpansionCore, and SubWord are called by the Key Function to complete this task.

The new expanded keys are stored in eleven static variables after the key function produces them. We created the key_decr_configuration function, whose job is to take the results of the KeyExpansion and stored it in the static variables. This function also resolves a conflict between read and write the expanded keys at the same clock cycle.

The directives used in the KeyFunction are the same as before. More specifically, the ***#pragma HLS inline off*** is being used so as to not be inlined upward into the calling function. The directive ***#pragma HLS pipeline*** is set to ***II = 1*** to keep the desired target of the pipeline to 1 clock cycle. Additionally, we use the ***#pragma HLS array_partition*** for the ***variable = expandedKey*** by ***complete*** to partition it completely into individual elements. The expandedKey is the array holding the results of the KeyExpansion function. Finally, the use of the ***#pragma HLS unroll*** is needed for the for-loop used for the separation of the expandedKey array into the eleven variables.

The KeyExpansion function uses the directive ***#pragma HLS inline off***, so as to not be inlined upward into the calling function. The rest of them (KeyExpansionCore, and SubWord) don't include any directives.

Core Function

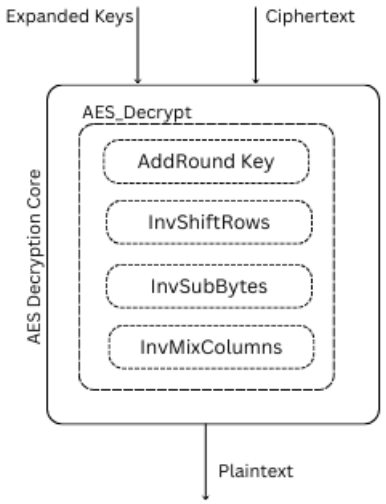


FIGURE 5.3: AES Decryption Core Function Block Diagram

The Core Function is the one that handles the incoming/outcoming data and calls the corresponding function to perform the wanted process. The function takes as input variables the input data from the ps2ip FIFO as well as the expanded keys calculated from the Key Function, and after produces its results sends them to the ip2ps fifo.

The function contains one FSM, that continuously reads if we have new incoming valid data, and calls the AES_Decrypt function to perform the decryption process. In this function the directive ***#pragma HLS pipeline II=1 enable_flush*** is being used, as to be able to process new data every one cycle, and if the data valid at the input of the pipeline goes inactive to flush and empty the pipeline.

Additionally, we set the directive ***#pragma HLS INTERFACE ap_ctrl_none port = return***. Function-level protocols, also called block-level I/O protocols, provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The ***ap_ctrl_none*** means that the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_-done) are not created. Block-level I/O protocols can be assigned to a port for the function ***return*** value.

It also contains the directive ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculations to be implemented in parallel.

AES_Decrypt

The AES_Decrypt is responsible for the proper use of the individual functions needed to perform the decryption process which is described in sec. 3.2.3.

Firstly, the directive ***#pragma HLS inline region*** is being used. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function.

Also, we include the directive ***#pragma HLS pipeline*** set to ***II = 1***. This means that they process new inputs every 1 clock cycle.

Finally, we use the ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculation to be implemented in parallel.

AddRoundKey

The function AddRoundKey is the same as in the AES Encryption, it uses the directive ***#pragma HLS inline off*** so as to not be inlined upward into the calling functions or regions. It also includes the directive ***#pragma HLS unroll*** to store the results in the state array in parallel.

InvMixColumns

The InvMixColumns function uses the directive ***#pragma HLS inline off***. This means that the functions should not be inlined upward into any calling functions or regions. It also uses the directive ***#pragma HLS unroll*** for storing its results in the state array in parallel.

InvSubBytes

The InvSubBytes function uses the directive ***#pragma HLS inline off***. This means that the function should not be inlined upward into any calling functions or regions. It also uses the directive ***#pragma HLS unroll*** for storing their results in the state array in parallel.

InvShiftRows

In this function, the directive ***#pragma HLS unroll*** is being used for parallel storing the results in the state array.

ps2ip fifo and ip2ps fifo

The ps2ip and the ip2ps are 64-depth FIFOs that keep the input and output data respectively. The ps2ip FIFO takes the input data and promotes them into the Core function, and the ip2ps FIFO takes the results of the Core function and sends them as output data.

5.1.1 Synthesis Report

The Synthesis Report of our AES Decryption project is shown in fig.5.4. As we can see, we have successfully achieved the Initiation Interval of 1, which means that the project processes new data and produces a new result every 1 clock cycle. Our final latency is 25, which indicates that we will get our first result after 25 clock cycles. Furthermore, the overall resource utilization is maintained to a minimum.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.30	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
25	25	1	1	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	0	-	0	20	-
Instance	110	-	15603	30768	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	99	-
Register	-	-	11	-	-
Total	110	0	15614	30937	0
Available	1824	2520	548160	274080	0
Utilization (%)	6	0	2	11	0

FIGURE 5.4: AES Decryption Synthesis Report

5.1.2 Directives Summary

In the following table, we summarized all the directives used in our AES Decryption project compared with the ones used in its corresponding of AES Encryption.

Directives	*AES Encryption*	*AES Decryption*
inline	✓	✓
allocation	✗	✗
unroll	✓	✓
array_map	✗	✗
array_partition	✓	✓
interface	✓	✓
pipeline	✓	✓
dataflow	✓	✓
stream	✓	✓

TABLE 5.1: Directives of AES Decryption

5.2 RC6 Encryption

The block diagram of the top module of the RC6 Encryption project is again demonstrated in fig. 5.1 .

rc6_top_module (Top Function)

The top-level function for the RC6 Encryption project is rc6_top_module. It has the same structure as the top function of our targeted framework. It takes the variables key and input_enc as inputs and output_enc as outputs. The directive ***# pragma HLS DATAFLOW*** is used in the function. The ***DATAFLOW*** pragma enables task-level pipelining, which allows functions and loops to overlap in their operation, enhancing RTL concurrency and overall design performance.

Again in this project, for all three of its input/output variables, the directive ***#INTERFACE axis register both port = <name>*** is being used. To signify that the functions' arguments are all implemented as AXI4-Stream interfaces. The directive ***#INTERFACE ap_ctrl_none port = return*** was also added to prevent the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_done) from being formed.

Due to the dataflow directive the use of the ***#pragma HLS STREAM variable=<variable> depth=<int> dim=<int>*** is needed. It was used to generate a FIFO for each of the function's three arguments. In particular, we have a 64-depth FIFO dimension of 1 for the input and output data and a 4-depth FIFO dimension of 1 for the key.

Finally, this function is responsible for the calls of the rest of our design functions, as well as the interconnections of their variables. Note that, all variables are type **stream <axiWord>**, where axiWord is a struct containing the variables ap_uint<128> data, ap_uint<16> strb, and ap_uint<1> last.

Check Key Function

The Check Key function is exactly the same as the one in the AES Encryption project, described in sec.4.2.3.

Key Function

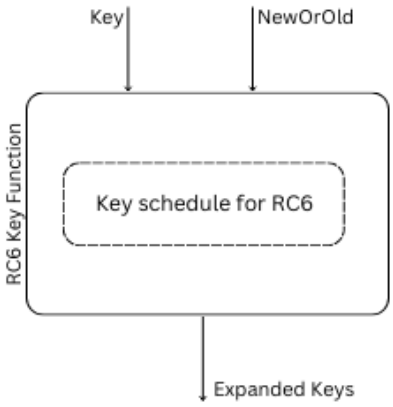


FIGURE 5.5: RC6 Key Function Block Diagram

The Key Function has a similar structure as the one in AES Encryption. Specifically, again in this case, it consists of two FSMs, the first one to continuous reading if we have new incoming valid keys due to the nature of the protocol our project is based on. The second one exists to determine, based on the result of the Check Key function, if the expanded keys needed for the encryption process are the ones we stored or we need to initiate the process to create new ones.

The directives ***#pragma HLS inline region*** is being used. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function, but not inlining recursively through the hierarchy.

The directive ***#pragma HLS pipeline*** is set to ***II = 1*** to keep the desired target of the pipeline to 1 clock cycle.

The code used for the creation of the expanded key is described in sec. 3.3.1, is incorporated in the body of this function. The use of the ***#pragma HLS unroll*** is needed for the for-loop that performs the calculations of the expanded keys as to be implemented in parallel. We also used the directive ***#pragma HLS dependence*** for the arrays S and L, set to type ***inter***, which indicates dependence between different loop iterations, and dependence to ***false***. This allows the HLS tool to perform operations in parallel if the function or loop is pipelined, or the loop is unrolled.

The expanded keys are stored in 44 32-bit variables.

Core Function

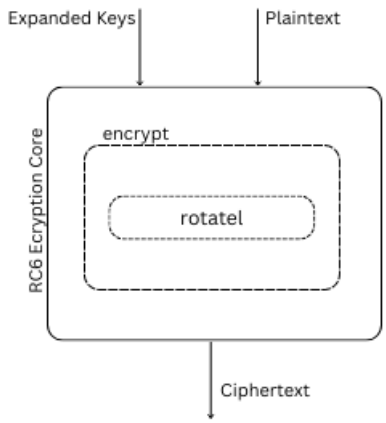


FIGURE 5.6: RC6 Encryption Core Function Block Diagram

The Core Function is similar to the initial framework. The function takes as input variables the input data from the ps2ip FIFO as well as the 44 expanded keys calculated from the Key Function, and after producing its results sends them to the ip2ps FIFO.

The function contains one FSM, that continuously reads if we have new incoming valid data, and calls the encrypt function to perform the encryption process of the RC6. In this function the directive ***#pragma HLS pipeline II=1 enable_flush*** is being used, as to be able to process new data every one cycle, and if the data valid at the input of the pipeline goes inactive to flush and empty the pipeline.

Additionally, we set the directive ***#pragma HLS INTERFACE ap_ctrl_none port = return***. Function-level protocols, also called block-level I/O protocols, provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The ***ap_ctrl_none*** means that the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_-done) are not created. Block-level I/O protocols can be assigned to a port for the function ***return*** value.

It also contains the directive ***#pragma HLS unroll*** directive for all the for-loops included in the code, to make all the calculations to be implemented in parallel.

encrypt

The encrypt function performs the encryption process of the RC6 algorithm described in sec. 3.3.2.

The directive ***#pragma HLS inline region*** is being used. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function.

Also, we include the directive ***#pragma HLS pipeline*** set to ***II = 1***. This means that they process new inputs every 1 clock cycle.

Finally, we use the ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculation to be implemented in parallel.

rotatel

The rotatel function performs the left rotation needed in the encryption process and the creation of the expanded keys. In this function, we use the ***#pragma HLS inline region*** directive. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function.

ps2ip fifo and ip2ps fifo

The ps2ip and the ip2ps are 64-depth FIFOs that keep the input and output data respectively. The ps2ip FIFO takes the input data and promotes them into the Core function, and the ip2ps FIFO takes the results of the Core function and sends them as output data.

5.2.1 Synthesis Report

The Synthesis Report of our RC6 Encryption project is shown in fig.5.7. As we can see, we have successfully achieved the Initiation Interval of 1, which means that the project processes new data and produces a new result every 1 clock cycle. Our final latency is 100, which indicates that we will get our first result after 100 clock cycles. Furthermore, the overall resource utilization is maintained at a low level, but not as much as the AES Encryption.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
100	100	1	1	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	362	-
FIFO	0	-	0	97	-
Instance	-	120	35419	69634	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	792	-
Register	-	-	88	-	-
Total	0	120	35507	70885	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	4	6	25	0

FIGURE 5.7: RC6 Encryption Synthesis Report

5.2.2 Directives Summary

In the following table, we summarized all the directives used in our RC6 Encryption project.

Directives	*RC6 Encryption*
inline	✓
allocation	✗
unroll	✓
array_map	✗
array_partition	✗
interface	✓
pipeline	✓
dataflow	✓
stream	✓
dependence	✓

TABLE 5.2: Directives of RC6 Encryption

5.3 RC6 Decryption

The block diagram of the top module of the RC6 Decryption project is again demonstrated in fig. 5.1 .

rc6_top_module (Top Function)

The top-level function for the RC6 Decryption project is rc6_top_module. It is the same as the RC6 Encryption, described in sec. 5.2 and it uses the same directives.

Check Key Function

The Check Key function is exactly the same as the one in the AES Encryption project, described in sec.4.2.3.

Key Function

The Key Function is the same as the one in RC6 Encryption, described in sec. 5.2.

Core Function

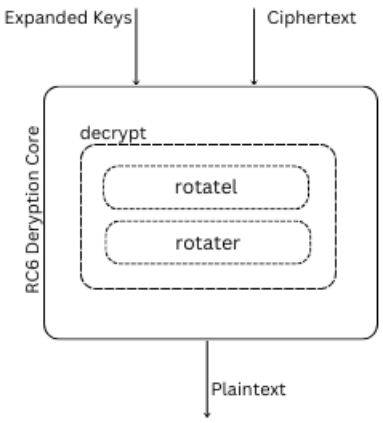


FIGURE 5.8: RC6 Decryption Core Function Block Diagram

The Core Function is the same as the one in the RC6 Encryption project, described in sec. 5.2, with the only difference that in this case the decrypt function is being called instead of the encrypt.

decrypt

The decrypt function performs the decryption process of the RC6 algorithm described in sec. 3.3.2.

The directive ***#pragma HLS inline region*** is being used. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function.

Also, we include the directive ***#pragma HLS pipeline*** set to ***II = 1***. This means that they process new inputs every 1 clock cycle.

Finally, we use the ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculation to be implemented in parallel.

rotatel

The rotatel function is the same as in the RC6 Encryption project, described in sec. 5.2.

rotater

The rotater function performs the right rotation needed in the decryption process of the RC6 algorithm. In this function, we use the ***#pragma HLS inline region*** directive. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function.

ps2ip fifo and ip2ps fifo

The ps2ip and the ip2ps are 64-depth FIFOs that keep the input and output data respectively. The ps2ip FIFO takes the input data and promotes them into the Core function, and the ip2ps FIFO takes the results of the Core function and sends them as output data.

5.3.1 Synthesis Report

The Synthesis Report of our RC6 Decryption project is shown in fig.5.7. As we can see, we have successfully achieved the Initiation Interval of 1, which means that the project processes new data and produces a new result every 1 clock cycle. Our final latency is 91, which indicates that we will get our first result after 91 clock cycles. Furthermore, the overall resource utilization is maintained at a low level, a little lower than the RC6 Encryption.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
91	91	1	1	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	362	-
FIFO	0	-	0	97	-
Instance	-	120	33795	66649	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	792	-
Register	-	-	88	-	-
Total	0	120	33883	67900	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	4	6	24	0

FIGURE 5.9: RC6 Decryption Synthesis Report

5.3.2 Directives Summary

In the following table, we summarized all the directives used in our RC6 Decryption project compared with the ones used in its corresponding RC6 Encryption.

Directives	*RC6 Encryption*	*RC6 Decryption*
inline	✓	✓
allocation	✗	✗
unroll	✓	✓
array_map	✗	✗
array_partition	✗	✗
interface	✓	✓
pipeline	✓	✓
dataflow	✓	✓
stream	✓	✓
dependence	✗	✓

TABLE 5.3: Directives of RC6 Decryption

5.4 Blowfish Encryption

The block diagram of the top module of the Blowfish Encryption project is again demonstrated in fig. 5.1.

blowfish (Top Function)

The top-level function for the Blowfish Encryption project is blowfish. It has the same structure as the top function of our targeted framework. It takes the variables key and data_in_enc as inputs and data_out_enc as outputs. The directive ***# pragma HLS DATAFLOW*** is used in the function. The ***DATAFLOW*** pragma enables task-level pipelining, which allows functions and loops to overlap in their operation, enhancing RTL concurrency and overall design performance.

For all three of its input/output variables, we use the directive ***#INTERFACE axis register both port = <name>***. To signify that the functions' arguments are all implemented as AXI4-Stream interfaces. The directive ***#INTERFACE ap_ctrl_none port = return*** was also added to prevent the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_done) from being formed.

Due to the dataflow directive the use of the ***#pragma HLS STREAM variable=<variable> depth=<int> dim=<int>*** is needed. It was used to generate a FIFO for each of the function's three arguments. In particular, we have a 64-depth FIFO dimension of 1 for the input and output data and a 4-depth FIFO dimension of 1 for the key.

Finally, this function is responsible for the calls of the rest of our design functions, as well as the interconnections of their variables. Note that, the input and output variables are type **stream <axiWord>**, where axiWord is a struct

containing the variables ap_uint<64> data, ap_uint<8> strb, and ap_uint<1> last, and the key is type **stream <axiKey>**, where axiKey is a struct containing the variables ap_uint<128> data, ap_uint<16> strb, and ap_uint<1> last.

Check Key Function

The Check Key function is exactly the same as the one in the AES Encryption project, described in sec.4.2.3.

Key Function

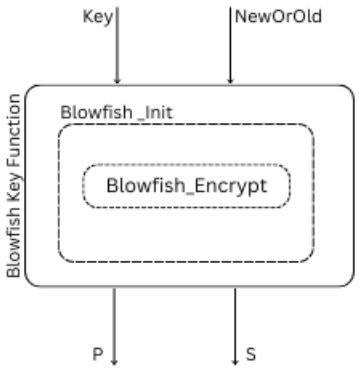


FIGURE 5.10: Blowfish Key Function Block Diagram

The Key Function has a similar structure as the ones described before. Specifically, again in this case, it consists of two FSMs, the first one to continuous reading if we have new incoming valid keys due to the nature of the protocol our project is based on. The second one exists to determine, based on the result of the Check Key function, if the subkeys needed for the encryption process are the ones we stored or if we need to initiate the process to create new ones.

The directive ***#pragma HLS inline region*** is being used. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function, but not inlining recursively through the hierarchy.

The directive ***#pragma HLS pipeline*** is set to ***II = 1*** to keep the desired target of the pipeline to 1 clock cycle.

When there is the need to generate the subkeys, the function Blowfish_Init is called to complete this task. The results of the P array are stored in a static array size 18, and the 4 S-boxes are stored in 4 static arrays size 256.

The use of the ***#pragma HLS unroll*** is needed for the for-loops that perform the calculations of the subkeys as to be implemented in parallel.

Blowfish_Init

The way the Blowfish_Init generated the subkeys is described in sec. 3.4.1. The directive ***#pragma HLS inline region*** is being used. This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function, but not inlining recursively through the hierarchy.

The directive ***#pragma HLS pipeline*** is set to ***II = 1*** to keep the desired target of the pipeline to 1 clock cycle.

The use of the ***#pragma HLS unroll*** is needed for the for-loops that perform the calculations of the subkeys as to be implemented in parallel.

Core Function

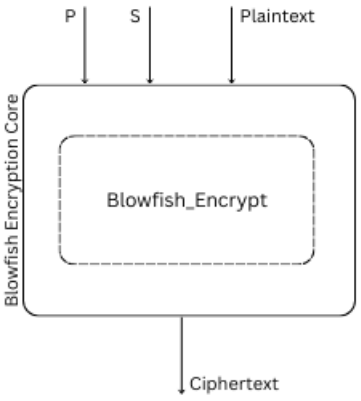


FIGURE 5.11: Blowfish Encryption Core Block Diagram

The Core Function is similar to the initial framework. The function takes as input variables the input data from the ps2ip FIFO as well as the P-array and the S-boxes calculated from the Key Function, and after produces its results sends them to the ip2ps fifo.

The function contains one FSM, that continuously reads if we have new incoming valid data, and calls the Blowfish_Encrypt function to perform the encryption process of the Blowfish algorithm. In this function the directive ***#pragma HLS pipeline II=1 enable_flush*** is being used, as to be able to process new data every one cycle and if the data valid at the input of the pipeline goes inactive to flush and empty the pipeline.

Additionally, we set the directive ***#pragma HLS INTERFACE ap_ctrl_none port = return***. Function-level protocols, also called block-level I/O protocols, provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The ***ap_ctrl_none*** means that the handshake signal ports (ap_start, ap_idle, ap_ready, and ap_-done) are not created. Block-level I/O protocols can be assigned to a port for the function ***return*** value.

Blowfish_Encrypt

The Blowfish_Encrypt function performs the encryption process of the Blowfish algorithm described in sec. 3.4.2.

The directive ***#pragma HLS inline off*** is being used. This means that the function should not be inlined upward into any calling functions or regions.

Also, we include the directive ***#pragma HLS pipeline II = 1*** which indicates that is set to process new inputs every 1 clock cycle.

Finally, we use the ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculation to be implemented in parallel.

ps2ip fifo and ip2ps fifo

The ps2ip and the ip2ps are 64-depth FIFOs that keep the input and output data respectively. The ps2ip FIFO takes the input data and promotes them into the Core function, and the ip2ps FIFO takes the results of the Core function and sends them as output data.

5.4.1 Synthesis Report

The Synthesis Report of our Blowfish Encryption project is shown in fig.5.12. Unfortunately, the Blowfish Encryption didn't perform as well as the other projects. Even though we adjust the algorithm to the targeted framework, the algorithm wasn't able to be pipelined with II=1 and produce results every clock cycle.

As we can see from the synthesis result, the interval has a range of value between 17 and 9753 clock cycles, meaning the fastest we can process new data is every 17 clock cycles and the slowest every 9753 clock cycles. The same phenomenon we can spot in the initial latency, which also has a range of values between 24 and 9775 clock cycles.

The cause of this is the complexity of the Blowfish_Init function. It performs a very large number of calculations, that have dependencies between their intermediate results and are unable to perform different operations in the same clock cycle. The same applies to the Blowfish_Encrypt function which also carries dependencies between its intermediate results.

The resource utilization is kept at a low level, not as low as the AES Encryption project, with the exception of the use of BRAMs, which is higher in the AES. But overall, the project didn't perform in the way we wanted.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.93	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
24	9775	17	9753	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	94	-
FIFO	0	-	0	27	-
Instance	10	-	17152	38727	-
Memory	16	-	0	0	-
Multiplexer	-	-	-	198	-
Register	-	-	22	-	-
Total	26	0	17174	39046	0
Available	1824	2520	548160	274080	0
Utilization (%)	1	0	3	14	0

FIGURE 5.12: Blowfish Encryption Synthesis Report

5.4.2 Directives Summary

In the following table we summarised all the directives used in our Blowfish Encryption project.

Directives	*Blowfish Encryption*
inline	✓
allocation	✗
unroll	✓
array_map	✗
array_partition	✗
interface	✓
pipeline	✓
dataflow	✓
stream	✓
dependence	✗

TABLE 5.4: Directives of Blowfish Encryption

5.5 Blowfish Decryption

The block diagram of the top module of the Blowfish Decryption project is again demonstrated in fig. 5.1.

blowfish (Top Function)

The top-level function for the Blowfish Decryption project is blowfish. It is the same as the Blowfish Encryption, described in sec. 5.4 and it uses the same directives.

Check Key Function

The Check Key function is exactly the same as the one in the AES Encryption project, described in sec.4.2.3.

Key Function

The Key Function is the same as the one in Blowfish Encryption, described in sec. 5.4.

Blowfish_Init Function

The Blowfish_Init is the same as the one in Blowfish Encryption, described in sec. 5.4.

Blowfish_Encrypt

The Blowfish_Encrypt is the same as the one in Blowfish Encryption, described in sec. 5.4. It is included in this project due to its use from the Blowfish_Init.

Core Function

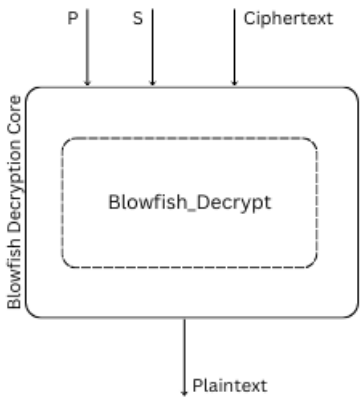


FIGURE 5.13: Blowfish Decryption Core Block Diagram

The Core Function is the same as the one in the Blowfish Encryption project, described in sec. 5.4, with the only difference that in this case the Blowfish_Decrypt function is being called instead of the Blowfish_Encrypt.

Blowfish_Decrypt

The Blowfish_Decrypt function performs the decryption process of the Blowfish algorithm described in sec. 3.4.3.

The directive ***#pragma HLS inline off*** is being used. This means that the function should not be inlined upward into any calling functions or regions.

Also, we include the directive ***#pragma HLS pipeline II = 1*** which indicates that is set to process new inputs every 1 clock cycle.

Finally, we use the ***#pragma HLS unroll*** directive for all the for-loops included in the code, as to make all calculation to be implemented in parallel.

ps2ip fifo and ip2ps fifo

The ps2ip and the ip2ps are 64-depth FIFOs that keep the input and output data respectively. The ps2ip FIFO takes the input data and promotes them into the Core function, and the ip2ps FIFO takes the results of the Core function and sends them as output data.

5.5.1 Synthesis Report

The Synthesis Report of our Blowfish Decryption project is shown in fig.5.14. As it was expected, the Blowfish Decryption project has a similar performance to the Blowfish Encryption project.

Also, in this case, the algorithm wasn't able to be pipelined with II=1 and produce results every one clock cycle. the interval has a range of value between 17 and 9753 clock cycles, meaning the fastest we can process new data is every 17 clock cycles and the slowest every 9753 clock cycles. The same phenomenon we can spot in the initial latency, which also has a range of values between 24 and 9775 clock cycles.

The cause again is the complexity of the calculation and the dependencies between their intermediate results of the Blowfish_Init, Blowfish_Encrypt, and Blowfish_Decrypt functions.

The resource utilization is kept at a low level, not as low as the AES Encryption project, with the exception of the use of BRAMs, which is higher in the AES. But overall, the project didn't perform in the way we wanted.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.93	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
24	9775	17	9753	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	94	-
FIFO	0	-	0	27	-
Instance	10	-	17465	38813	-
Memory	16	-	0	0	-
Multiplexer	-	-	-	198	-
Register	-	-	22	-	-
Total	26	0	17487	39132	0
Available	1824	2520	548160	274080	0
Utilization (%)	1	0	3	14	0

FIGURE 5.14: Blowfish Decryption Synthesis Report

5.5.2 Directives Summary

In the following table, we summarized all the directives used in our Blowfish Decryption project compared with the ones used in its corresponding Blowfish Encryption.

Directives	***Blowfish Encryption***	***Blowfish Decryption***
inline	✓	✓
allocation	✗	✗
unroll	✓	✓
array_map	✗	✗
array_partition	✗	✗
interface	✓	✓
pipeline	✓	✓
dataflow	✓	✓
stream	✓	✓
dependence	✗	✗

TABLE 5.5: Directives of Blowfish Decryption

5.6 Summary

In the following table, we summarize all directives we met during the implementation of each algorithm and display which ones are being used in each of our six projects.

Directives	*AES Encrypt*	*AES Decrypt*	*RC6 Encrypt*	*RC6 Decrypt*	*Blowfish Encrypt*	*Blowfish Decrypt*
inline	✓	✓	✓	✓	✓	✓
allocation	✗	✗	✗	✗	✗	✗
unroll	✓	✓	✓	✓	✓	✓
array_map	✗	✗	✗	✗	✗	✗
array_partition	✓	✓	✗	✗	✗	✗
interface	✓	✓	✓	✓	✓	✓
pipeline	✓	✓	✓	✓	✓	✓
dataflow	✓	✓	✓	✓	✓	✓
stream	✓	✓	✓	✓	✓	✓
dependence	✗	✗	✓	✓	✗	✗

TABLE 5.6: Directives of All Algorithms

The conclusions considering the usage of directives are the following:

- As we can tell, for each algorithm, the encryption and decryption part consist of the same directives since they have the same structure.
- In the case of RC6, the majority of directives are the same as the ones in our targeted framework of AES Encryption, except for the ***array_partition***. The RC6 doesn't need the array_partition directive, since the separation of the array holding the expanded keys was performed by hand (the expanded keys are kept in 44 variables).
- For RC6 we needed to include the ***dependence*** directive to distinguish the dependence of its arrays between different loop iterations during the process of its key function.
- In the case of Blowfish, the majority of directives are again the same as the ones in our targeted framework of AES Encryption, except for the ***array_partition***. We tried to include this directive in the Blowfish encryption and decryption project for its four S-boxes but didn't end up in the final form, because Vivado was unable to finish the synthesis of the project due to lack of memory.
- The ***allocation*** directive wasn't used in the final form of any algorithm, since it was more resource utilization effective to allow the Vivado to

decide automatically the number of instances of the functions needed in each algorithm.

- Again, the ***array_map*** directive was not used in any algorithm's final version. The reason for this is that either the algorithm did not have any arrays that could be combined into a larger one (in the case of RC6), or during the optimization process we reduced the array to the absolute necessary ones and there was no need to combine those into larger arrays or make any difference in the number of BRAMs used (in case of AES and Blowfish).

Chapter 6

Design Verification and Performance Evaluation from Actual Runs

In this chapter, we will demonstrate the results of our hardware implementations. More specifically, we analyze the performance of AES, RC6, and Blowfish, both encryption and decryption, based on a series of simulation-level tests. Furthermore, we verified our design by executing a series of runs in real hardware and checking the validity of the outcomes from each dataset we used.

6.1 FPGA Platforms

Initially, all of our designs were tested and implemented using Xilinx's Vivado HLS 2017.1 with the Zynq UltraScale+ ZCU102 Evaluation Platform as the target device (xczu9eg-ffvb1156-1-i-es1). We ran a series of simulation-level tests with various sizes of streaming data to evaluate the behavior of each algorithm in terms of area utilization, latency, interval, and how the clock value can affect them, and their throughput.

Later on, we wanted to evaluate our architecture's performance on a physical FPGA Board. For this purpose, we used the PYNQ-Z1 (Python Productivity for Zynq-7000 ARM/FPGA SoC). With some small adjustments in the design of our two AES projects (Encryption and Decryption), we successfully download them on the board and evaluate their overall performance compared to a software implementation.

6.1.1 Zynq UltraScale+ ZCU102 Evaluation Platform

The ZCU102 is a high-performance, high-speed hardware/software design platform providing the integration of hardware, software, IP, and reference designs which enables quicker time-to-innovation for researchers. Based around the Xilinx Zynq Ultrascale+ MPSoC, it is equipped with industry-standard peripherals, connects, and interfaces that offer a rich set of features suitable for a wide range of applications.

System Logic Cells	Flip-Flops	LUTs	Block RAM (Mb)	DSP Slices
599,550	548,160	274,080	912	2,520

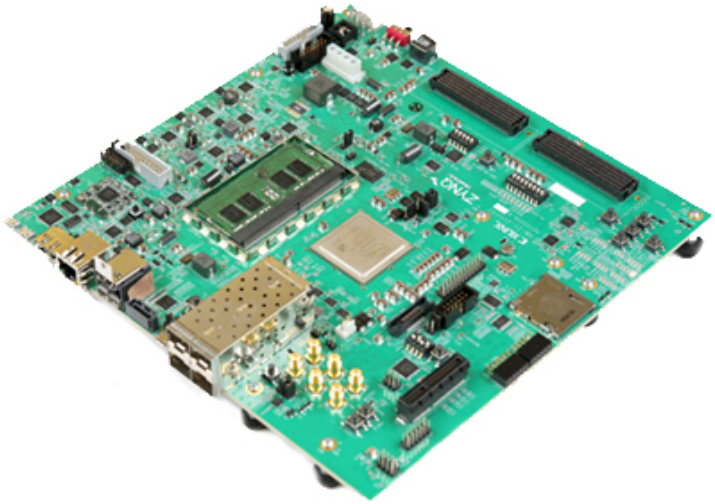


FIGURE 6.1: Zynq UltraScale+ ZCU102 Evaluation Platform

6.1.2 PYNQ

Digilent PYNQ-Z1 Python Productivity Board for Zynq-7000 ARM/FPGA SoC is a general-purpose, programmable platform for embedded systems. The PYNQ-Z1 Board is designed to be used with PYNQ. PYNQ is an open-source framework that enables embedded programmers to explore the capabilities of Xilinx Zynq All Programmable SoCs (APSoCs) without having to design programmable logic circuits. Alternately, the APSoC is programmed using Python and the code is developed and tested directly on the PYNQ-Z1.

The programmable logic circuits are imported as hardware libraries and programmed through their APIs. The PYNQ-Z1 board is the hardware platform for the PYNQ open-source framework.

The PYNQ-Z1 supports multi-media applications with onboard audio and video interfaces. The Board is designed to be easily extensible with Pmod, Arduino, and Grove peripherals, as well as general-purpose IO pins. The PYNQ-Z1 Board can be also expanded with USB peripherals including WiFi, Bluetooth, and Webcams.

Logic slices	Flip-Flops	LUTs	Block RAM (KB)	DSP Slices
13,300	106,400	53,200	630	220

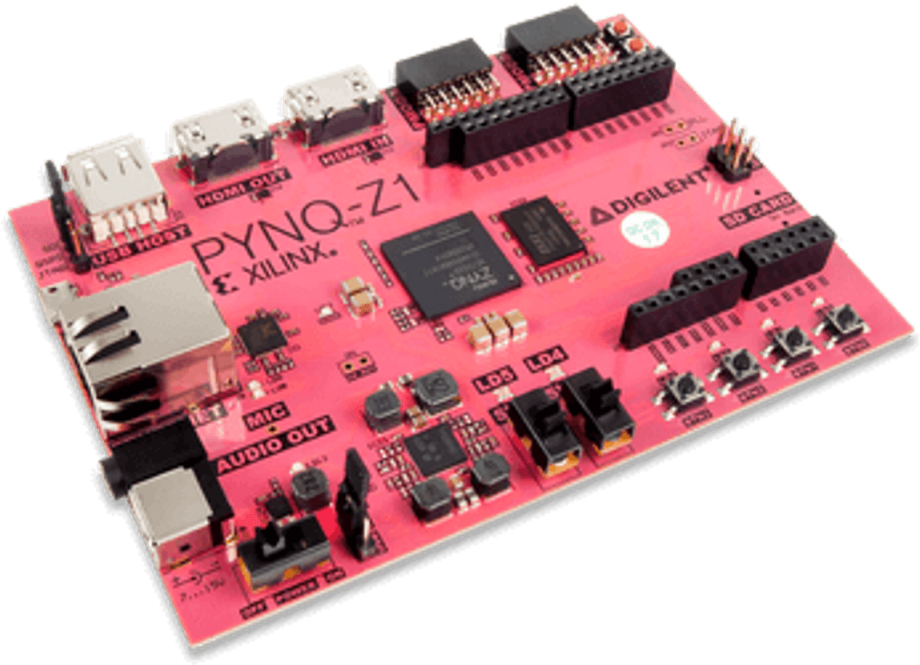


FIGURE 6.2: PYNQ Z1 Board

6.2 Throughput

Throughput is the flow rate. It is the rate at which the number of units goes through the process per unit of time. In our point of view, throughput is defined as the amount of data that is successfully transmitted through a system in a certain amount of time, measured in bits per second (bps).

Throughput can be calculated using the following formula:

$$T = \frac{I}{F} \tag{6.1}$$

where:

T = Throughput
I = Inventory (the number of units in the production process - in our case the data packets that are being encrypted/decrypted)
F = The time the inventory units spend in production from start to finish

6.3 Simulation Testing and Results

We tested each algorithm using **30 packets** of streaming data and changing the algorithm's key every **10 packets** to evaluate its behavior. The term **packet** refers to a group of 128-bit variables for AES and RC6, or 64-bit variables for Blowfish, with the last variable of each packet signaling the end of the stream. We began with a size of a 64-byte packet, and we doubled its size for each test until we reached 1 MB.

6.3.1 RTL Waveform

The following image shows the RTL waveform provided by the Vivado IDE, which shows the behavior of the AES Encryption design. More specifically, it demonstrates the output of the algorithm for the first packet of data it processed.

As we can see, the first expected result should be available at 415 ns, unfortunately, we don't have a valid result, instead, we get an "XXXXX...XX" value.

The "XXXX..XX" value means unknown/impossible to determine this value/result. The problem that is causing this, originates in the **key function**. During the process of producing the outcome, the key functions give to the

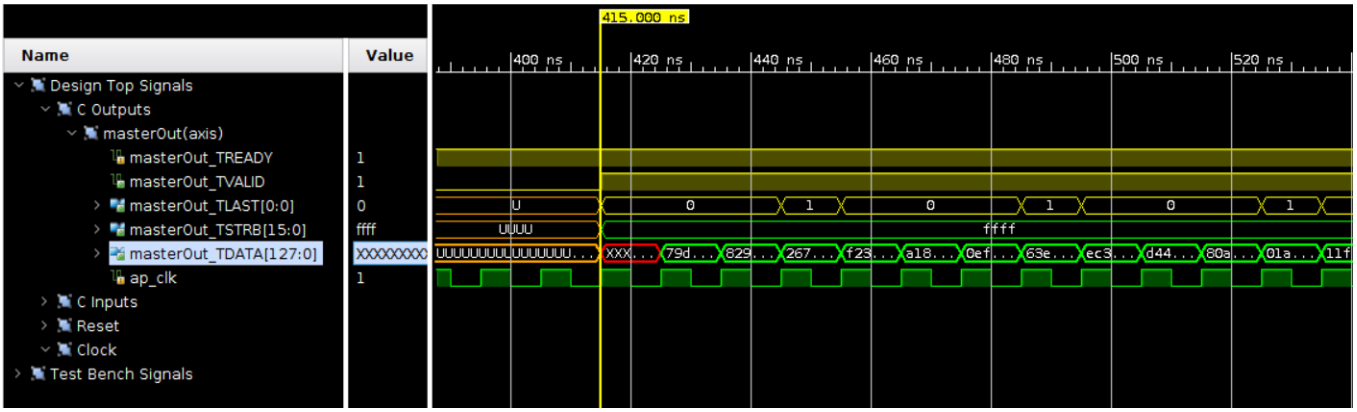


FIGURE 6.3: first encrypted output

encryption function the variables that hold the expanded keys one clock cycle earlier before they are ready, and that is causing the invalid result.

This happens every time we have a new key. Due to this problem, we have an invalid encrypted result of the first 128 bits of plaintext, but all the following outcomes are correct.

We tried to fix this issue, but unfortunately, we didn't succeed. The Vivado HLS compiler is creating this problem during the translation from the C++ code to VHDL. Cause of the directive **#pragma HLS pipeline**, the tool is being ordered to produce a code that will provide us with a result every 1 clock cycle after the initial latency, and during the translations to VHDL, to meet our command, the expanded keys variables are being used before the initialization with the correct values. Every control we added to prevent this issue was ignored in the translation.

A possible solution to this is if we intervene in the VHDL code and add control and prevent the algorithm from using the expanded keys before they are ready.

All of the algorithms behave in the same way. The first outcome when the key changes is invalid, and the rest are valid. Usually in the process of encryption/decryption of data, the key doesn't change very often, thereafter the loss can be considered negligence, since we only lose 128-bits or 64-bits of data depending on the algorithm.

6.3.2 Simulations Run Time Results

In the following tables, we gathered the run times of each algorithm for its first **valid** result as well as its last, for all the individual packet sizes.

AES Simulation Run Time Results

The 6.1 table shows the run times for AES Encryption and Decryption.

	Size of Stream Packet (bytes)	Total Amount of Data Send (bytes)	AES Encryption first result (ns)	AES Encryption last result (ns)	AES Decryption first result (ns)	AES Decryption last result (ns)
1	64	1920	425	1615	425	1615
2	128	3840	425	2815	425	2815
3	256	7680	425	5215	425	5215
4	512	15360	425	10015	425	10015
5	1024	30720	425	19615	425	196155
6	2048	61440	425	38915	425	38915
7	4096	122880	425	77215	425	77215
8	8192	245760	425	154015	425	154015
9	16384	491520	425	307615	425	307615
10	32768	983040	425	614815	425	614815
11	65536	1966080	425	1229215	425	1229215
12	131072	3932160	425	2458015	425	2458015
13	262144	7864320	425	4915615	425	4915615
14	524288	15728640	425	9830815	425	9830815
15	1048576	31457280	425	19661215	425	19661215

TABLE 6.1: AES Simulation Run Time Results

As we can see, the time that we got the first correct result of the first packet regardless of the packet size is stable.

In detail, as mentioned earlier each packet of AES consists of a specific number of 128-bit variables, based on its size. More specifically, the 64-byte packet consists of 4 128-bit variables that are being transmitted through the algorithm, and we send 30 packets, as a result, we have a total amount of 120 transmissions.

As we can see, we get the first correct encrypted result in 425 ns, the second one in 435 ns, the third in 445 ns,..., etc. Meaning that after the initial latency, and the first outcome being ignored due to invalid results, we get a new correct result every 10 ns which translates in every clock cycle.

The same behavior we observe in every case regardless of the packet size. Every time the key changes (in our case every 10 packets) we lose the first

outcome of the first packet due to an invalid result, and after that, we receive a correct result every clock cycle.

RC6 Simulation Run Time Results

The 6.2 table shows the results for RC6 Encryption and Decryption.

	Size of Stream Packet (bytes)	Total Amount of Data Send (bytes)	RC6 Encryption first result (ns)	RC6 Encryption last result (ns)	RC6 Decryption first result (ns)	RC6 Decryption last result (ns)
1	64	1920	1175	2365	1085	2275
2	128	3840	1175	3565	1085	3475
3	256	7680	1175	5965	1085	5875
4	512	15360	1175	10765	1085	10675
5	1024	30720	1175	20365	1085	20275
6	2048	61440	1175	39565	1085	39475
7	4096	122880	1175	77965	1085	77875
8	8192	245760	1175	154765	1085	154675
9	16384	491520	1175	308375	1085	308275
10	32768	983040	1175	615565	1085	615475
11	65536	1966080	1175	1229965	1085	1229875
12	131072	3932160	1175	2458765	1085	2458675
13	262144	7864320	1175	4916365	1085	4916275
14	524288	15728640	1175	9831565	1085	9831475
15	1048576	31457280	1175	19661965	1085	19661875

TABLE 6.2: RC6 Simulation Run Time Results

Based on the results of the RC6 Encryption and Decryption algorithms, we can state that the RC6 has a similar behavior as the AES. Again the time that we got the first correct result of the first packet for every packet size is stable. For RC6 Encryption and Decryption, we lose the first result when the key changes as well, and after that we have a valid result every clock cycle.

Blowfish Simulation Run Time Results

The 6.3 table shows the results for Blowfish Encryption and Decryption.

	Size of Stream Packet (bytes)	Total Amount of Data Send (bytes)	Blowfish Encryption first result (ns)	Blowfish Encryption last result (ns)	Blowfish Decryption first result (ns)	Blowfish Decryption last result (ns)
1	64	1920	99035	602195	99035	602195
2	128	3840	99035	914195	99035	914195
3	256	7680	99035	1538195	99035	1538195
4	512	15360	99035	2786195	99035	2786195
5	1024	30720	99035	75282195	99035	5282195
6	2048	61440	99035	10274195	99035	10274195
7	4096	122880	99035	20258195	99035	20258195
8	8192	245760	99035	40226195	99035	40226195
9	16384	491520	99035	80162195	99035	80162195
10	32768	983040	n/a	n/a	n/a	n/a
11	65536	1966080	n/a	n/a	n/a	n/a
12	131072	3932160	n/a	n/a	n/a	n/a
13	262144	7864320	n/a	n/a	n/a	n/a
14	524288	15728640	n/a	n/a	n/a	n/a
15	1048576	31457280	n/a	n/a	n/a	n/a

TABLE 6.3: Blowfish Simulation Run Time Results

Regarding the results of the Blowfish Encryption and Decryption, it is important to mention that we only manage to get the results for packet sizes up to 16 KBytes, after that Vivado HLS was unable to produce the results due to lack of memory.

Both cases of Blowfish produce new outcomes every 1300 ns (130 clock cycles). When we have a new key the algorithms give us the first outcome after 97530 ns. The additional delay is created by the computational complexity of the key function.

Once again the time for the first result of the first packet is stable in both cases, regardless of the size of the packet.

Finally, Blowfish produces an invalid result for the first outcome when the key changes, similarly to the rest algorithms.

6.4 Throughput Results

In the 6.4 table, we are demonstrating the throughput of each algorithm for every size of stream packet, which is calculated based on the mathematical formula 6.1.

	Total Amount of Data Send (bits)	AES Encr	AES Decr	RC6 Encr	RC6 Decr	Blowfish Encr	Blowfish Decr
1	15360	9.5108	9.5108	6.4947	6.7516	0.0255	0.0255
2	30720	10.913	10.913	8.6171	8.8403	0.0336	0.0336
3	61440	11.7814	11.7814	10.3001	10.4579	0.0399	0.0399
4	122880	12.2696	12.2696	11.4148	11.511	0.0441	0.0441
5	245760	12.5292	12.5292	12.0678	12.1213	0.0465	0.0465
6	491520	12.6306	12.6306	12.4231	12.4514	0.0478	0.0478
7	983040	12.7312	12.7312	12.6087	12.6233	0.0485	0.0485
8	1966080	12.7655	12.7655	12.7036	12.711	0.0489	0.0489
9	3932160	12.7827	12.7827	12.7512	12.7554	0.0491	0.0491
10	7864320	12.7914	12.7914	12.7758	12.7776	n/a	n/a
11	15728640	12.7957	12.7957	12.7879	12.7888	n/a	n/a
12	31457280	12.7978	12.7978	12.7939	12.7944	n/a	n/a
13	62914560	12.7989	12.7989	12.797	12.7972	n/a	n/a
14	125829120	12.7995	12.7995	12.7985	12.7986	n/a	n/a
15	251658240	12.7997	12.7997	12.7992	12.7993	n/a	n/a

TABLE 6.4: Throughput (Gbits/sec)

As seen in the following table 6.4 and the diagram 6.4, the throughput of AES and RC6 for encryption and decryption tends to stabilize at 12.79 Gbps as the amount of data being processed via the algorithms reaches 15 Mbits or higher.

The lower values of throughput numbers we observe for the smaller quantities of data are caused by the initial latency. If we increase the number of packets we send for processing in the cases of the smaller packet sizes and thus the total amount of bits, we will notice that the value of throughput will tend to stabilize at 12.79 Gbps as well because the initial latency won't have a big impact on it.

Furthermore, Blowfish Encryption and Description's throughput fall behind in comparison to the throughput of the other algorithms. The best value we got was 0.0491 Gbps. As demonstrated in 6.5, Blowfish Encryption and Description's throughput improves as the amount of data increases, but their overall performance is far from our target, which is at least 10 Gbps since Blowfish was unable to provide us with an outcome every clock cycle, as the others did.

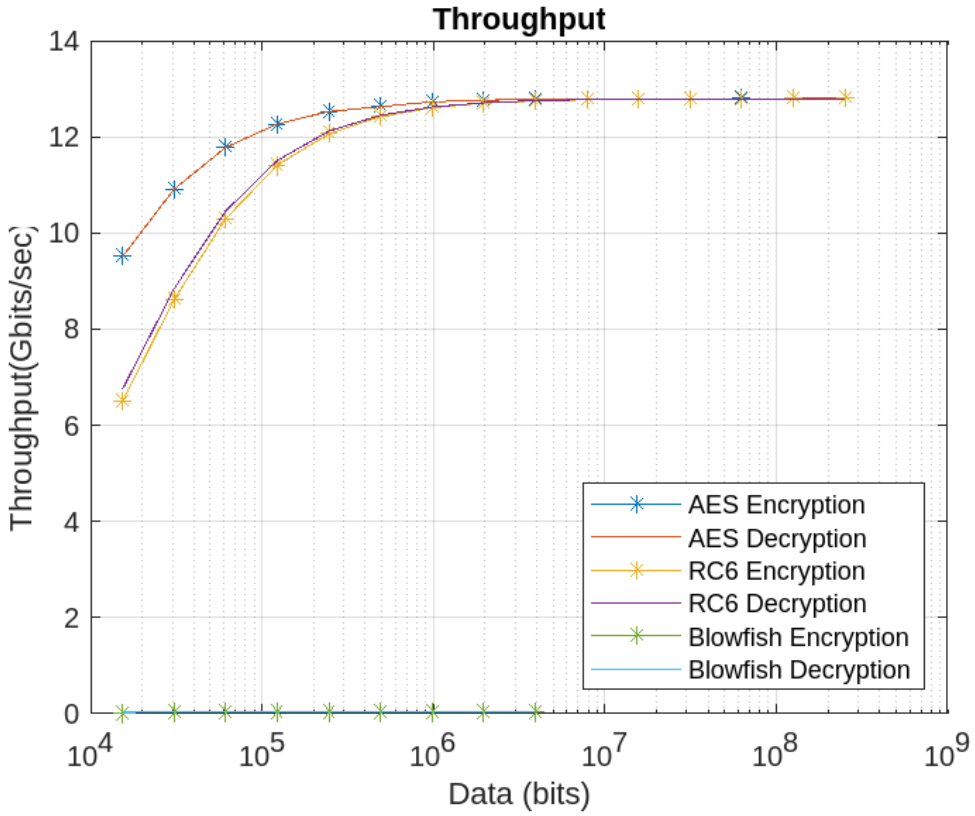


FIGURE 6.4: Throughput

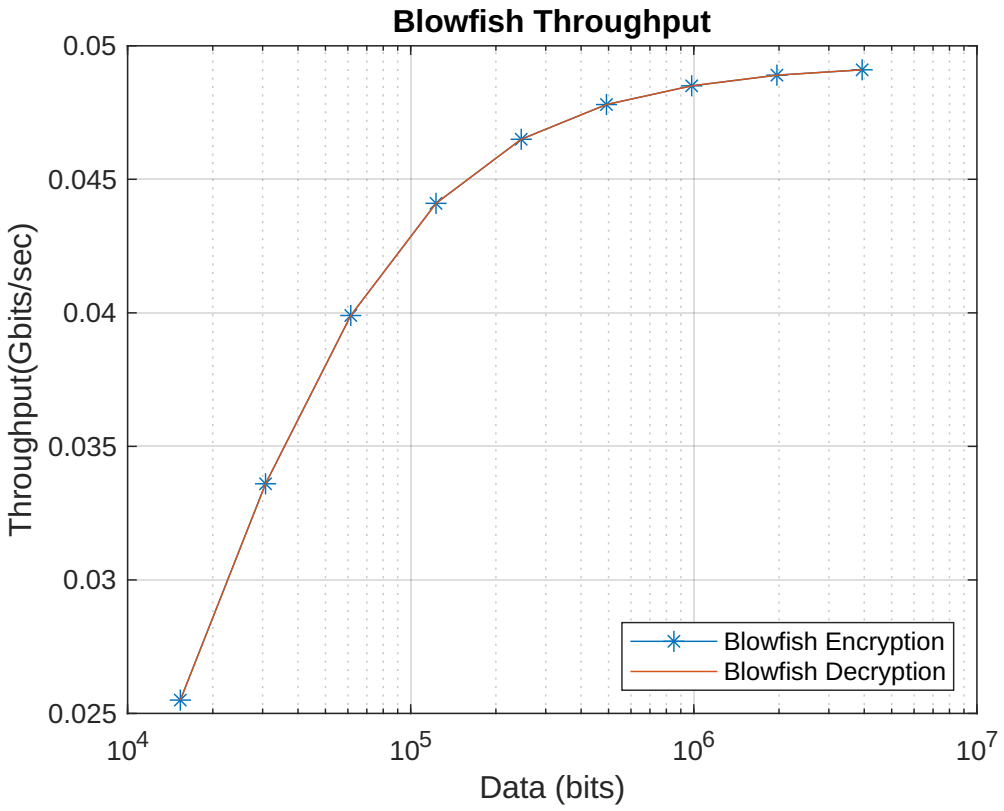


FIGURE 6.5: Throughput of Blowfish

6.5 Zynq UltraScale+ ZCU102 Resource Utilizations

The following table shows the resource utilization for all algorithms on the Zynq UltraScale+ ZCU102 Evaluation Platform as provided by Vivado HLS 2017.1.

Algorithm	**BRAM_18K**	**DSP**	**FF**	**LUT**
Available	1824	2520	548160	274080
AES Encryption	110	0	15614	14701
utilization %	6%	0	2%	2%
AES Decryption	110	0	15614	30937
utilization %	6%	0	2%	11%
RC6 Encryption	0	120	35507	70885
utilization %	0	4%	6%	25%
RC6 Decryption	0	120	33883	70885
utilization %	0	4%	6%	25%
Blowfish Encryption	26	0	30148	43660
utilization %	1%	0	5%	15%
Blowfish Decryption	26	0	30148	43660
utilization %	1%	0	5%	15%

TABLE 6.5: Algorithms Resource Utilization in Zynq UltraScale+ ZCU102

Based on the information above, AES Encryption has the lowest resource utilization of all the algorithms, and AES Decryption comes right after. The algorithms that occupied the most resources are the RC6 Encryption and Decryption.

6.6 Clocks and Resources

At this point, our goal was to observe how the clock's value can affect the usage of the available resources, as well as if it can influence the initial latency or the pipeline (interval) of each algorithm.

6.6.1 AES

Initially, the target clock for each algorithm was set at 10 ns as a default. In the case of AES, both Encryption and Decryption, the estimated clock was calculated at 2.3 ns with 1.25 ns uncertainty. As we lower the value of the target clock to get closer to its estimated, we can observe the following:

AES Encryption									
Clock			Latency		Interval		Resources		
Target	Est.	Unc.	min	max	min	max	BRAM	FF	LUT
10	2.3	1.25	25	25	1	1	110 (6%)	15614 (2%)	14701 (2%)
5	2.3	0.63	25	25	1	1	110 (6%)	15614 (2%)	14701 (2%)
3	2.3	0.38	25	25	1	1	110 (6%)	15614 (2%)	14701 (2%)
2.68	2.3	0.34	25	25	1	1	110 (6%)	15614 (2%)	14701 (2%)
2.5	1.67	0.31	27	27	1	1	110 (6%)	18579 (3%)	14815 (5%)
2	1.67	0.25	27	27	1	1	110 (6%)	18579 (3%)	14815 (5%)
1.5	1.35	0.19	55	55	1	1	110 (6%)	20019 (3%)	16038 (5%)
1	1.35	0.13	56	56	1	1	110 (6%)	22836 (4%)	16137 (5%)

TABLE 6.6: AES Encryption Clock and Resources Behaviour

AES Decryption									
Clock			Latency		Interval		Resources		
Target	Est.	Unc.	min	max	min	max	BRAM	FF	LUT
10	2.3	1.25	25	25	1	1	110 (6%)	15614 (2%)	30937 (11%)
5	2.3	0.63	25	25	1	1	110 (6%)	15614 (2%)	30937 (11%)
3	2.3	0.38	25	25	1	1	110 (6%)	15614 (2%)	30937 (11%)
2.68	2.3	0.34	25	25	1	1	110 (6%)	15614 (2%)	30937 (11%)
2.5	1.92	0.31	27	27	1	1	110 (6%)	18579 (3%)	31051 (11%)
2	1.67	0.25	36	36	1	1	110 (6%)	19856 (3%)	31218 (11%)
1.5	1.35	0.19	55	55	1	1	110 (6%)	20019 (3%)	32274 (11%)
1	1.35	0.13	56	56	1	1	110 (6%)	22836 (4%)	32373 (11%)

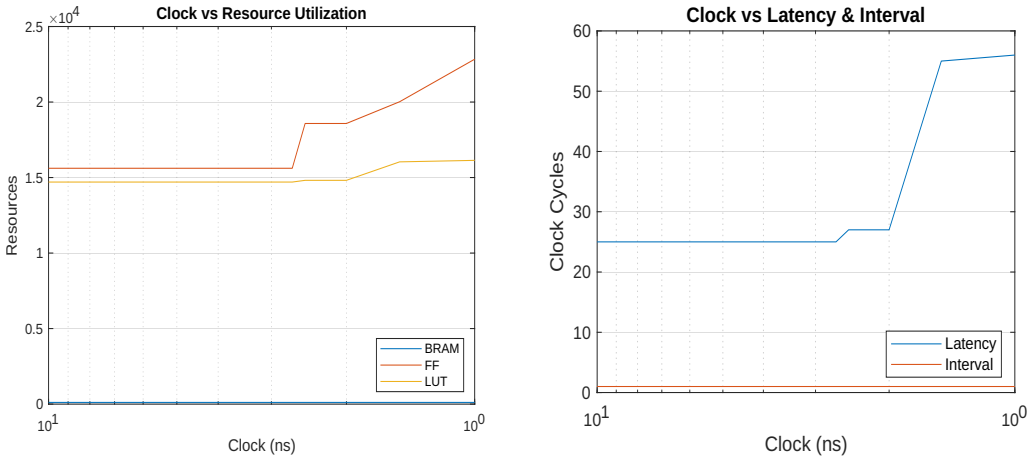
TABLE 6.7: AES Decryption Clock and Resources Behaviour

1. The lower value of the clock we got, without any changes in the latency or the resource utilization was 2.68 ns.

2. The Interval remained constant at 1 clock cycle, regardless of the value of the target clock.

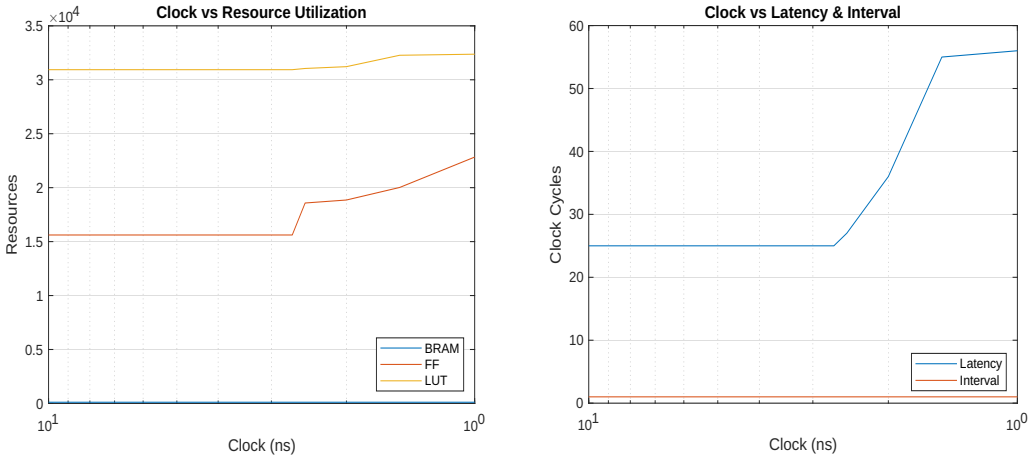
3. The number of BRAMs that are being used remained unchanged as well, regardless of the value of the target clock.

4. When we get lower to 2.68 ns we observe an increase in the initial latency, as well as in the number of Flip - Flops and LUTs that are being used.



(A) Resource Utilization Behaviour (B) Latency and Interval Behaviour

FIGURE 6.6: AES Encryption Resources, Latency, and Interval Behaviour



(A) Resource Utilization Behaviour (B) Latency and Interval Behaviour

FIGURE 6.7: AES Decryption Resources, Latency, and Interval Behaviour

6.6.2 RC6

Again, in the case of the RC6 algorithm, the target clock was set at 10 ns as a default. In both the Encryption and Decryption of RC6, the estimated clock was calculated at 8.75 ns with 1.25 ns uncertainty. As we lower the value of the target clock to get closer to its estimated, we can observe the following:

1. Any try to lower the value of the target clock below 10 ns, increase the initial latency, as well as the Flip-Flops and LUTs usage. Initially, as the clock value remained close to 10 ns, the increase in resources was quite small, of the order of 1%, but as the limit for the clock was pushed to reach a significantly smaller value, the resource usage was up around 3% to 4%. Additionally,

RC6 Encryption										
Clock			Latency		Interval		Resources			
Target	Est.	Unc.	min	max	min	max	BRAM	DCP	FF	LUT
10	8.75	1.25	100	100	1	1	0	120 (4%)	35507 (6%)	70885 (25%)
9	7.86	1.13	110	110	1	1	0	120 (4%)	38559 (7%)	68722 (25%)
8.75	7.49	1.09	123	123	1	1	0	120 (4%)	41051 (7%)	79809 (29%)
8	6.99	1	123	123	1	1	0	120 (4%)	41181 (7%)	81616 (29%)
5	4.28	0.63	214	214	1	1	0	160 (6%)	58058 (10%)	91376 (33%)

TABLE 6.8: RC6 Encryption Clock and Resources Behaviour

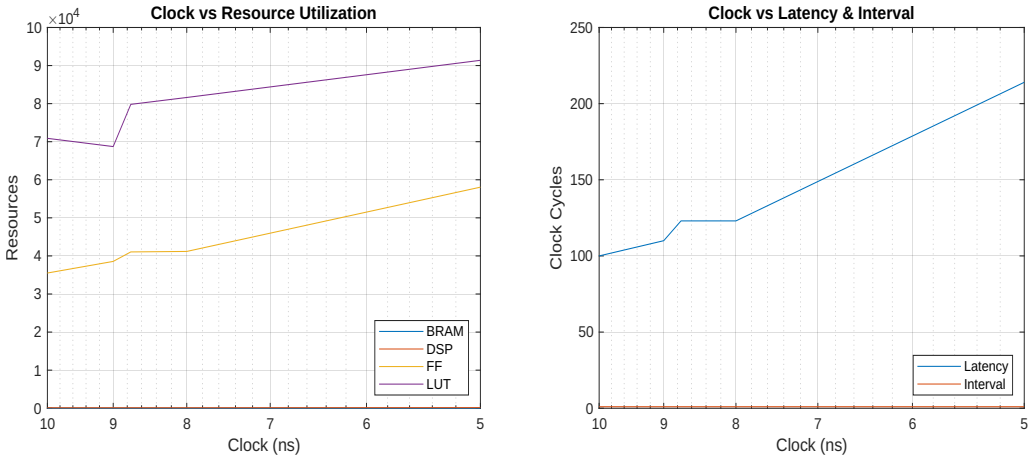
RC6 Decryption										
Clock			Latency		Interval		Resources			
Target	Est.	Unc.	min	max	min	max	BRAM	DCP	FF	LUT
10	8.75	1.25	91	91	1	1	0	120 (4%)	33883 (6%)	70885 (25%)
9	7.86	1.13	110	110	1	1	0	120 (4%)	40727 (7%)	69245 (25%)
8.75	7.49	1.09	123	123	1	1	0	120 (4%)	42035 (7%)	73506 (26%)
8	6.99	1	123	123	1	1	0	120 (4%)	42165 (7%)	75312 (27%)
5	4.28	0.63	214	214	1	1	0	160 (6%)	60866 (11%)	91376 (33%)

TABLE 6.9: RC6 Decryption Clock and Resources Behaviour

the latency had similar behavior, it started with small increases, around 10 to 20 clock cycles, and as the limit for the clock was pushed to 5 ns, we saw it double in size.

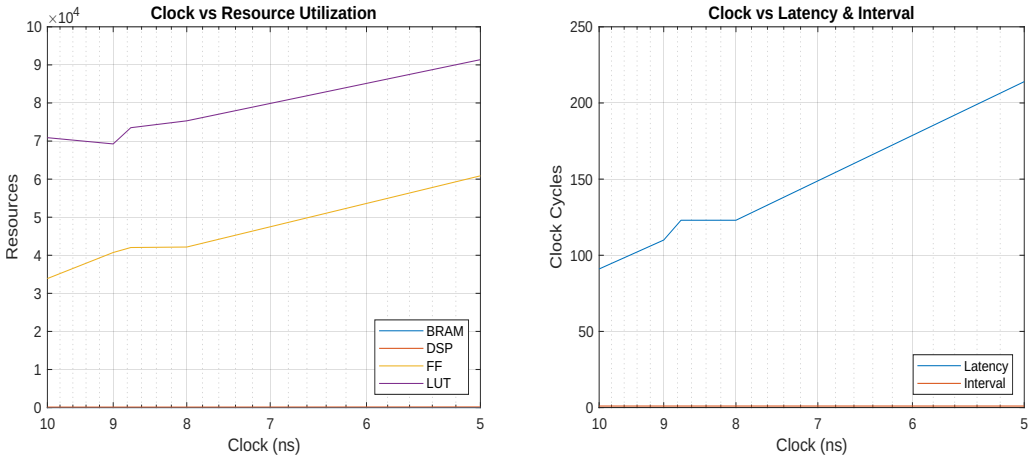
2. The Interval remained constant at 1 clock cycle, regardless of the value of the target clock.

3. The number of DSPs that are being used remained unchanged for most of the cases, only when we tried to push the clock at 5 ns we saw an increase in the order of 2% in the number of DSPs that are being used.



(A) Resource Utilization Behaviour (B) Latency and Interval Behaviour

FIGURE 6.8: RC6 Encryption Resources, Latency, and Interval Behaviour



(A) Resource Utilization Behaviour (B) Clock vs. Latency and Interval

FIGURE 6.9: RC6 Decryption Resources, Latency, and Interval Behaviour

6.6.3 Blowfish

For the Blowfish algorithm, the target clock was set at 10 ns as a default, as well. In both the Encryption and Decryption of Blowfish, the estimated clock was calculated at 6.16 ns with 1.25 ns uncertainty. As we lower the value of the target clock to get closer to its estimated, we can observe the following:

1. The number of Flip-Flops and LUTS that are being used, remained unchanged in most tries. Only when we set the target clock at 5 ns we saw an increase in the order of 1% to 2% of the recourse usage.

2. The maximum value of the Latency and Interval remained constant at 9775 and 9753 clock cycles, respectively, for most of the tries, except for the case of

Blowfish Encryption										
Clock			Latency		Interval		Resources			
Target	Est.	Unc.	min	max	min	max	BRAM	DCP	FF	LUT
10	6.16	1.25	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
7.5	6.16	0.94	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
7	6.12	0.88	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
6	5.25	0.75	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
5	4.66	0.62	24	17590	17	17568	26 (1%)	0	38752 (7%)	43907 (16%)

TABLE 6.10: Blowfish Encryption Clock and Resources Behaviour

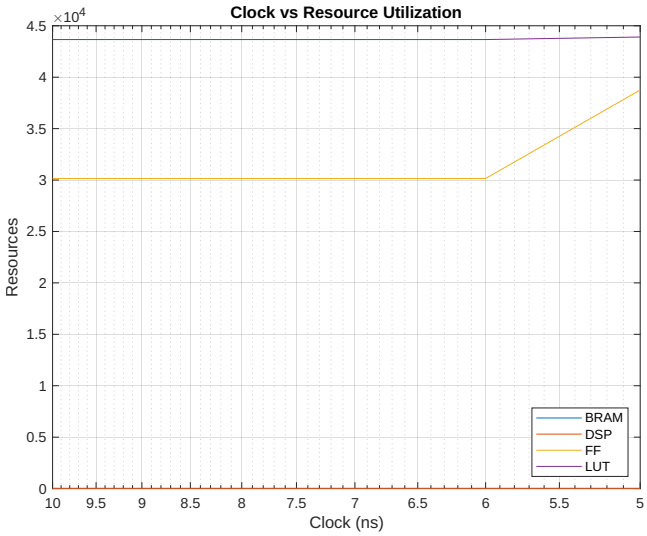
Blowfish Decryption										
Clock			Latency		Interval		Resources			
Target	Est.	Unc.	min	max	min	max	BRAM	DCP	FF	LUT
10	6.16	1.25	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
7.5	6.16	0.94	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
7	6.12	0.88	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
6	5.25	0.75	24	9775	17	9753	26 (1%)	0	30148(5%)	43660 (15%)
5	4.66	0.62	24	17590	17	17568	26 (1%)	0	38752 (7%)	43907 (16%)

TABLE 6.11: Blowfish Decryption Clock and Resources Behaviour

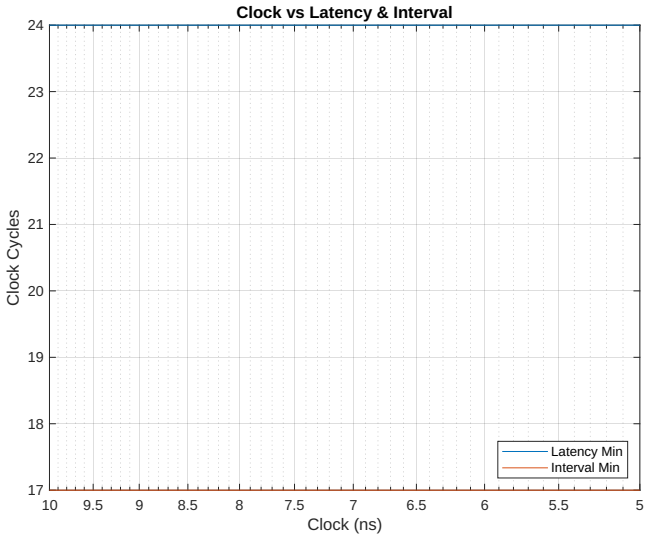
the 5 ns clock when their value almost doubled in size.

3. The minimal value of the Latency and Interval remained constant at 24 and 17 clock cycles, respectively, regardless of the value of the target clock.

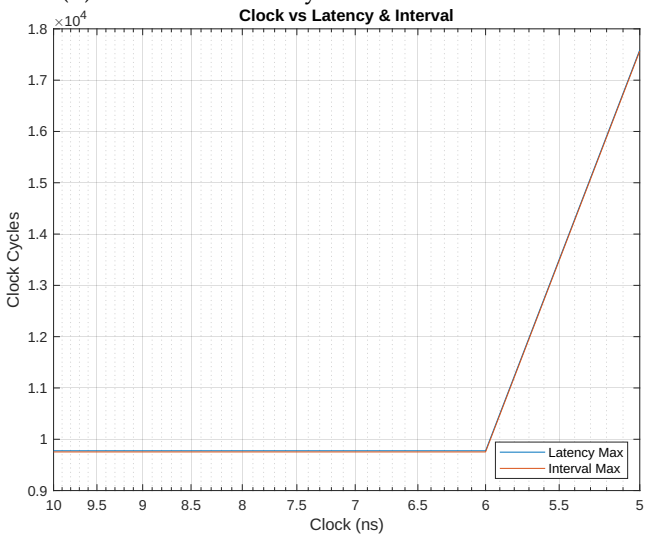
4. The number of BRAMs that are being used remained unchanged as well, regardless of the value of the target clock.



(A) Resource Utilization Behaviour

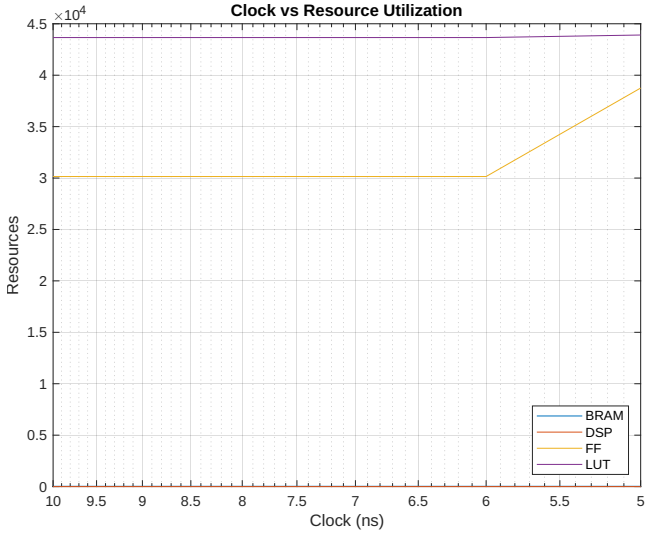


(B) Minimum Latency and Interval Behaviour

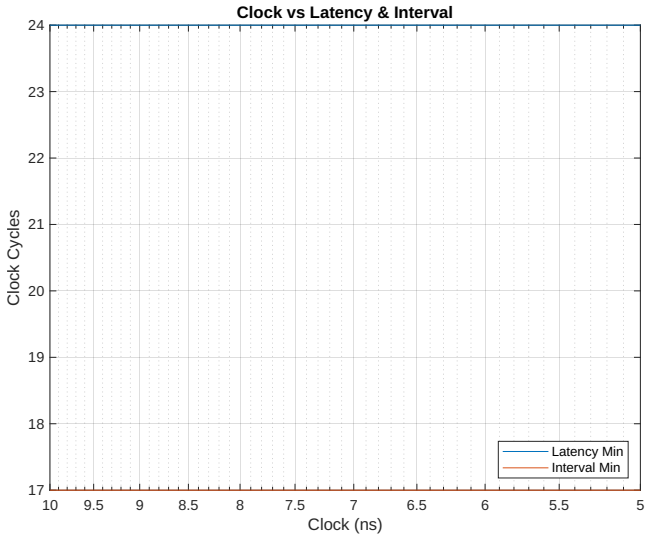


(C) Maximum Latency and Interval Behaviour

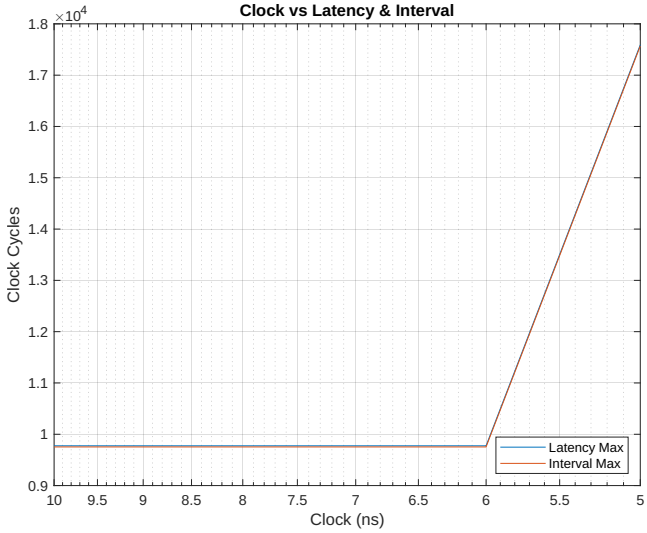
FIGURE 6.10: Blowfish Encryption Resources, Latency, and Interval Behaviour



(A) Resource Utilization Behaviour



(B) Minimum Latency and Interval Behaviour



(C) Maximum Latency and Interval Behaviour

FIGURE 6.11: Blowfish Decryption Resources, Latency, and Interval Behaviour

6.7 PYNQ Results

For the evaluation on a physical board of our architecture, we chose to use AES, both encryption and decryption designs. For this task, due to restrictions from the PYNQ Z1 FPGA board, we had to make some changes in the design. More specifically, due to the fact that the PYNQ Z1 provides us with 4 high-performance AXI3 slave ports, all the inputs/outputs of our design had to go through those 4 ports.

Initially, AES Encryption/Decryption had 2 128-bit inputs, the key, the plaintext/ciphertext, and 1 128-bit output, the ciphertext/plaintext. We change that to 2 64-bit inputs, that carry the plaintext/ciphertext, and 2 64-bit outputs to carry the result of the individual algorithm, ciphertext/plaintext. For the key, each algorithm uses the first value from both of the 64-bit inputs. Also, since all the data for processing are provided from DMAs, to signal the algorithm of the end of the steaming packet, the last value of the two 64-bit inputs is set to 0xFFFFFFFFFFFFFFFF.

6.7.1 PYNQ Synthesis Result

In this section, we present the resource utilization for the PYNQ Z1 board as provided by Vivavo HLS 2020.1.

AES ENCRYPTION	**BRAM_18K**	**DSP**	**FF**	**LUT**
Total	110	0	17005	21668
Available	280	220	106400	53200
Utilization (%)	39	0	15	40

TABLE 6.12: AES Encryption PYNQ Resource Utilization

AES DECRYPTION	**BRAM_18K**	**DSP**	**FF**	**LUT**
Total	110	0	44683	48915
Available	280	220	106400	53200
Utilization (%)	39	0	41	91

TABLE 6.13: AES Decryption PYNQ Resource Utilization

The first thing we can observe is that the design of the Decryption uses more than double the amount of Flip-Flops and LUTs in comparison to the design of the Encryption.

This happens due to the complexity of the calculations in the InvMixColumns Transformations. Even though we used multiplications with Galois Field to avoid the use of arrays for these calculations, the InvMixColumns Transformations are more complex compared to the MixColumns Transformations of the Encryption.

From this, we came to the conclusion that we are able to fit 2 modules of AES Encryption, in the PYNQ Board, instead of 1 module of AES Decryption due to resource usage.

6.7.2 PYNQ RTL Waveform

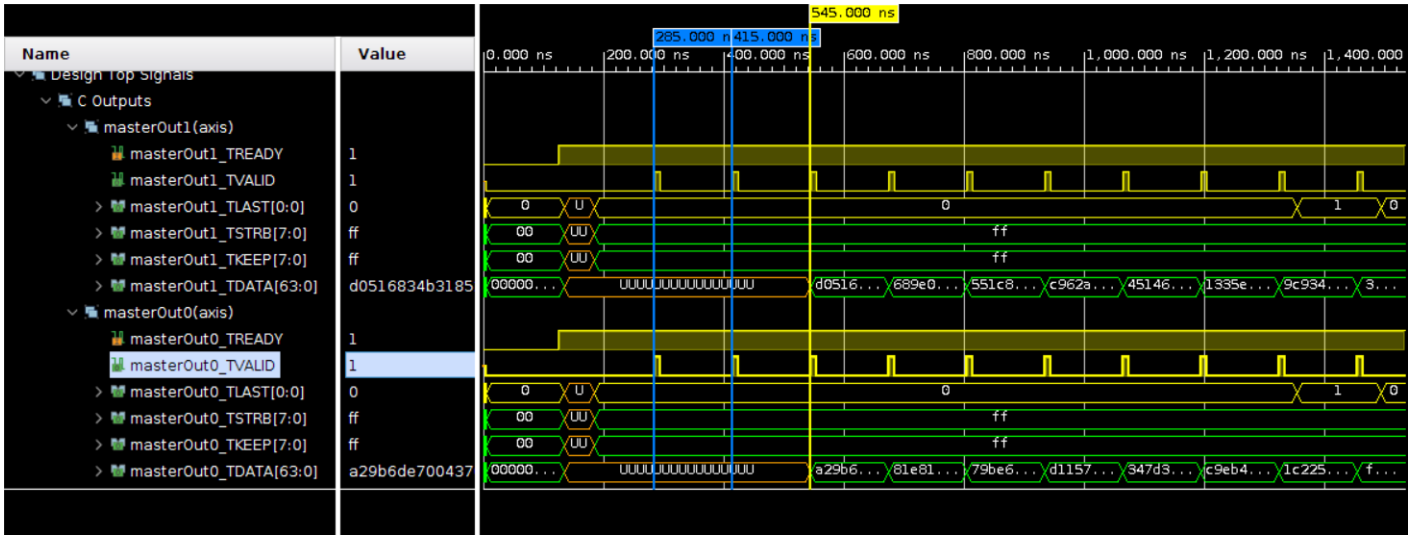


FIGURE 6.12: AES Encryption PYNQ output

In the case of AES ENCRYPTION, the first **two** results are **incorrect**, and the rest of them **correct**. The first value is the result of the encryption of the value that is being used as a key from the algorithm, and we can ignore it. Meaning that our design produces an invalid result for the first input, and the following are valid results.

Also, as we can see the run time for every result differs from the next one for 130 ns, and since the clock is set at 10 ns, this means that the algorithm produces a new outcome every 13 clock cycles.

In the case of AES DECRYPTION, the **first** result IS **incorrect**, and the rest of them **correct**. The first value is the result of the decryption of the value that is being used as a key from the algorithm, and we can ignore it. Meaning that our design produces valid results.

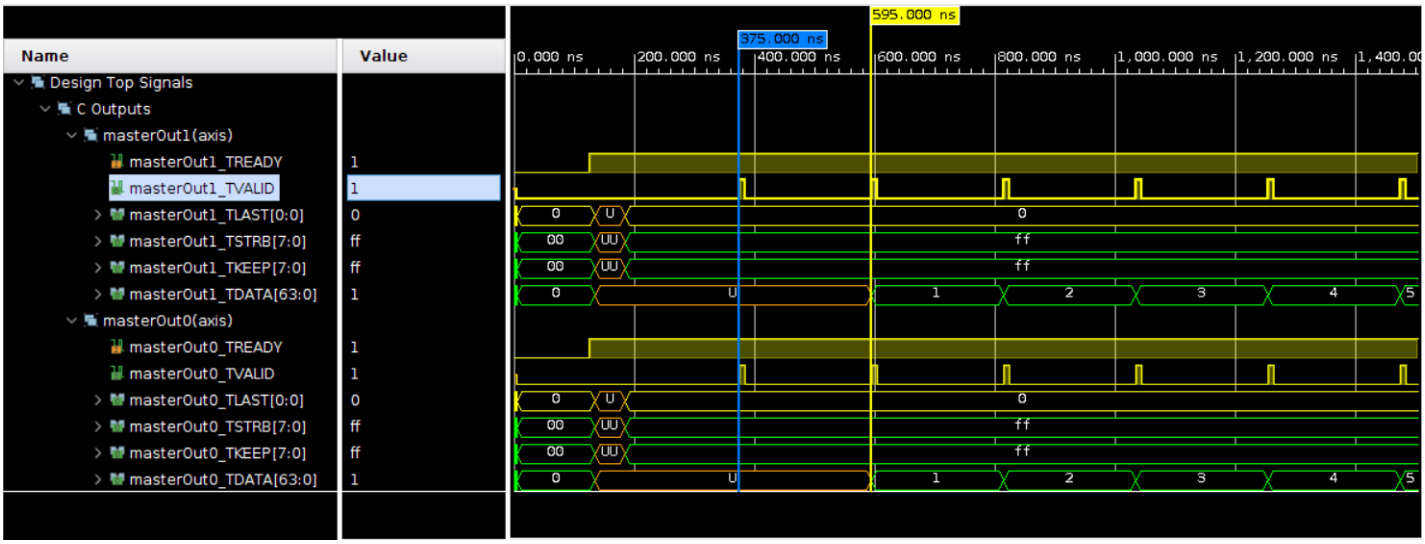


FIGURE 6.13: AES Decryption PYNQ output

Also, in the case of AES Decryption the run time for every result differs from the next one for 220 ns, and since the clock is set at 10 ns, this means that the algorithm produces a new outcome every 22 clock cycles.

6.7.3 Validation of the Algorithm

First of all, we needed to validate our design and figure out if the results of the encryption/decryption process are correct. To do so, initially, we created a small dataset with 10 128-bit values, a specific 128-bit key, and the termination value. We send those values to be encrypted and later use those results in the decryption design.

The results were consistent with the simulations. The Encryption loses two values, with the one being the key, and the Decryption loses only the value of the key, as mentioned earlier. The rest are correct.

After that, we performed the Known Answer Test (KAT) in type Variable Text, from The Advanced Encryption Standard Algorithm Validation Suite (AESAVS), Appendix D, section D.1 [36].

This test contains a specific value for the key and 128 different plaintext values and the corresponding values of ciphertext. Using those data, we create a dataset for each design and initialize the DMAs with those values. The results of this process were as expected. Once again, Encryption misses two values, with one being the key, and Decryption misses only the key. All the remaining are the same as the values provided in the test.

6.7.4 Hardware vs. Software Performance

For this section, we created a number of datasets of various sizes, for testing the performance of our hardware designs and their equivalent versions in software.

The initial C code of AES Encryption and Decryption we used can be found in [37]. We made some changes in the code to accommodate our hardware architecture, more specifically, AES in ECB mode, with a block size of 128 bits and a 128-bit key. Read and write in a file, so we can use the same datasets we used for the hardware designs.

The runs in software were performed in a high-speed server (Kronos) with the following specifications:

- Model name: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
- Architecture: x86_64
- CPU(s): 40
- Memory: 263850120 kB

with GCC -O3 compiler, using a single thread.

6.7.4.1 Run Time Results

In the following tables, we gathered the run times of our algorithms for each dataset in software and hardware.

AES Encryption					
No	**File Name**	**Data Size**	**HW Run Time (sec)**	**Server SW Run Time (sec)**	**Performance HW vs. SW**
1	dataset0	10	0.142	0.001	0.01
2	dataset1	128	0.145	0.002	0.01
3	dataset2	1000	0.149	0.006	0.04
4	dataset3	10000	0.17	0.058	0.34
5	dataset4	100000	0.3995	0.314	0.79
6	dataset5	200000	0.651	0.682	1.05
7	dataset6	300000	0.9035	0.927	1.03
8	dataset7	400000	1.03	1.082	1.05
9	dataset8	500000	1.419	1.517	1.07

TABLE 6.14: AES Encryption

Based on the results in the tables above we can observe the following:

AES Decryption					
No	**File Name**	**Data Size**	**HW Run Time (sec)**	**Server SW Run Time (sec)**	**Performance HW vs. SW**
1	dataset0	10	0.291	0.002	0.01
2	dataset1	128	0.293	0.007	0.02
3	dataset2	1000	0.36	0.037	0.10
4	dataset3	10000	0.412	0.22	0.53
5	dataset4	100000	1.03	1.626	1.58
6	dataset5	200000	1.663	3.232	1.94
7	dataset6	300000	2.323	4.846	2.09
8	dataset7	400000	2.962	6.404	2.16
9	dataset8	500000	3.671	7.978	2.17

TABLE 6.15: AES Decryption

1. The AES Encryption Hardware is 2 to 2.5 times faster than the AES Decryption Hardware. This happens due to the fact that the encryption consists of 2 independent and parallel modules of our design, and the decryption only consists of 1 module.

2. The AES Encryption Software is up to 5 times faster than the AES Decryption Software, especially in the bigger dataset. This happens cause, even though the process of encryption and decryption consists of similar steps, the calculations in decryption are higher in terms of complexity compared to those in encryption.

3. The performance of AES Encryption Hardware is similar to its Software performance as the amount of data that is being processed through the algorithm increases. As we can see, considering that the PYNQ board is small, low cost, and with low power consumption, with the proper design can successfully compete against a high-performance server.

For the smaller datasets hardware is far worst than software. This happens because the initial latency in hardware makes a bigger impact on the run times for smaller datasets, in contrast to the larger ones.

4. AES Decryption Hardware in comparison to its Software is up to 2 times better, especially for larger amounts of data. In this case, even though the software runs were executed on a high-speed server, the PYNQ board succeeded to execute faster the complex calculations of the decryption process.

6.7.4.2 Throughput Hardware vs. Software

In this section, we calculated the throughput of AES Encryption and Decryption based on the run times on the PYNQ board and the run times on the server.

No	File Name	Data Size	Total Amount of Data (bits)	HW AES Encryption	SW AES Encryption	HW AES Decryption	SW AES Decryption
1	dataset0	10	1280	0.009	1.28	0.0044	0.64
2	dataset1	128	16384	0.113	8.192	0.056	2.341
3	dataset2	1000	128000	0.859	21.333	0.356	3.459
4	dataset3	10000	1280000	7.529	22.069	3.107	5.818
5	dataset4	100000	12800000	32.04	40.764	12.427	7.872
6	dataset5	200000	25600000	39.324	37.537	15.394	7.921
7	dataset6	300000	38400000	42.501	41.424	16.53	7.924
8	dataset7	400000	51200000	49.709	47.32	17.286	7.995
9	dataset8	500000	64000000	45.102	42.189	17.434	8.022

TABLE 6.16: Throughput of HW and SW (Mbits/sec)

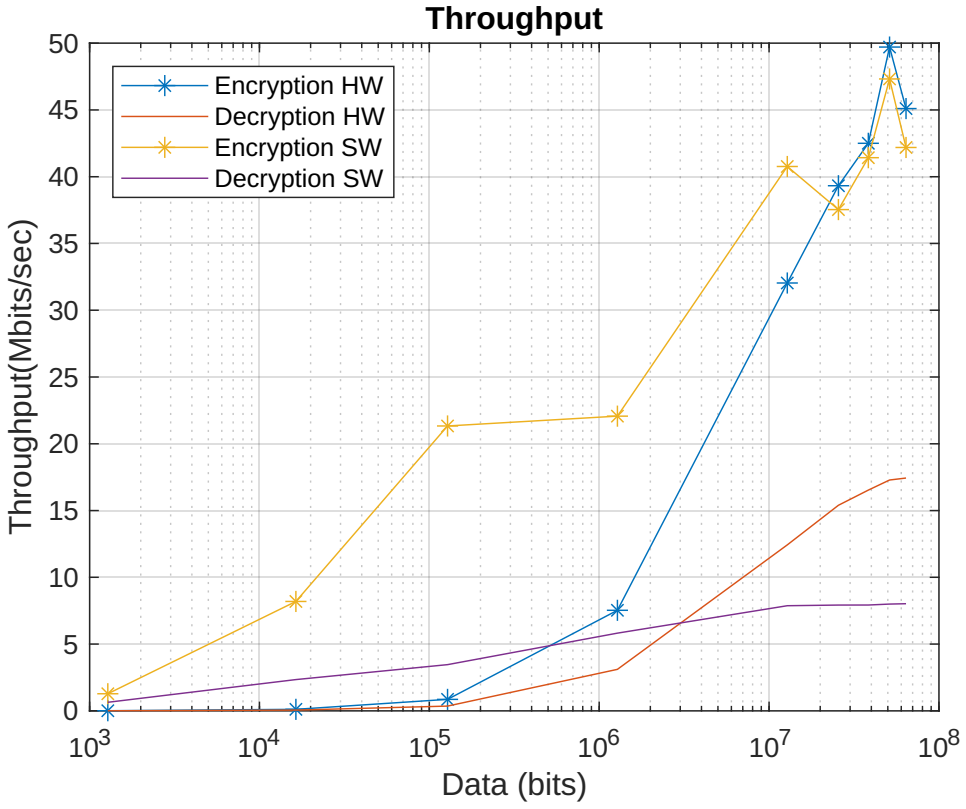


FIGURE 6.14: Throughput of HW in PYNQ and SW

Based on the 6.14 diagram, we can see that Encryption's throughput, in both cases of hardware and software, is better than the Decryption's throughput. But even though Encryption performance is better, the best value of throughput we got from its hardware design is 49.709 Mbps, which still is 257 times

worst than the throughput of AES Encryption's design in the Zynq UltraScale+ ZCU102, at 12,79 Gbps, we analyzed at sec 6.4

This makes us come to the conclusion that when we have the ability to work with expensive equipment, such as Zynq UltraScale+ ZCU102, gives us the flexibility to design a high-performance project since we have fewer restrictions to keep in mind.

But in a more realistic perspective, we aren't always able to have access to such expensive equipment. So by using low-cost equipment, such as the PYNQ-Z1 board may have an impact on the performance, but as we saw we can create a design that can compete with a high-speed server.

Chapter 7

Conclusions and Future Work

In the final chapter of this thesis, we summarize and evaluate our work. Additionally, we include some ideas for future work with the hope to inspire people to further research this subject.

7.1 Conclusions

The purpose of this thesis was the creation of hardware-accelerated cryptography IP that can be incorporated into a disaggregated datacenter.

The Advanced Encryption Standard even though it was created in 2001, and can be considered "old", can successfully stand in modern problems. Our proposed design of AES, when implemented in the Zynq UltraScale+ ZCU102, achieved a throughput of 12.79 Gbps, while the resource utilization was kept at a very low level.

Furthermore, our design when implemented at a small, low-cost, low-power consumption FPGA board, such as PYNQ-Z1, even though it loses in terms of performance against the Zynq UltraScale+ ZCU102, can perform as well, in case of encryption, or even two times better, in the case of decryption, against a high-speed server.

7.2 Future Work

Regarding future work, based on the performance of the AES-128, the other versions of AES can be implemented, and compare their performance to this work.

Additionally, some next-generation cryptography algorithms, such as Adiantum, can be explored if can be incorporated into the design of a disaggregated datacenter.

Finally, a hardware-accelerated IP can be produced by combining cryptography methods that will swap during the encryption process, increasing the level of security, and making it harder to breach through.

Appendix A

AES Specifications

A.1 Definitions

A.1.1 Glossary of Terms and Acronyms

The following definitions are used throughout this standard:

AES Advanced Encryption Standard.

Affine Transformation A transformation consisting of multiplication by a matrix followed by the addition of a vector.

Array An enumerated collection of identical entities (e.g., an array of bytes).

Bit A binary digit having a value of 0 or 1.

Block Sequence of binary bits that comprise the input, output, State, andRound Key. The length of a sequence is the number of bits it contains. Blocks are also interpreted as arrays of bytes.

Byte A group of eight bits that is treated either as a single entity or as an array of 8 individual bits.

Cipher Series of transformations that converts plaintext to ciphertext using the Cipher Key.

Cipher Key Secret, cryptographic key that is used by the Key Expansion routine to generate a set of Round Keys; can be pictured as a rectangular array of bytes, having four rows and Nk columns.

Ciphertext Data output from the Cipher or input to the Inverse Cipher.

Inverse Cipher Series of transformations that converts ciphertext to plaintext using the Cipher Key.

Key Expansion Routine used to generate a series of Round Keys from the Cipher Key.

Plaintext Data input to the Cipher or output from the Inverse Cipher.

Round Key Round keys are values derived from the Cipher Key using the Key Expansion routine; they are applied to the State in the Cipher and Inverse Cipher.

State Intermediate Cipher result that can be pictured as a rectangular array of bytes, having four rows and Nb columns.

S-box Non-linear substitution table used in several byte substitution transformations and in the Key Expansion routine to perform a one-for-one substitution of a byte value.

Word A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.

A.1.2 Algorithm Parameters, Symbols, and Functions

The following algorithm parameters, symbols, and functions are used throughout this standard :

AddRoundKey() Transformation in the Cipher and Inverse Cipher in which a Round Key is added to the State using an XOR operation. The length of a Round Key equals the size of the State (i.e., for Nb = 4, the Round Key length equals 128 bits/16 bytes).

InvMixColumns() Transformation in the Inverse Cipher that is the inverse of MixColumns().

InvShiftRows() Transformation in the Inverse Cipher that is the inverse of ShiftRows().

InvSubBytes() Transformation in the Inverse Cipher that is the inverse of SubBytes().

K Cipher Key.

MixColumns() Transformation in the Cipher that takes all of the columns of the State and mixes their data (independently of one another) to produce new columns.

Nb Number of columns (32-bit words) comprising the State. For this standard, Nb = 4.

Nk Number of 32-bit words comprising the Cipher Key. For this standard, Nk = 4, 6, or 8.

Nr Number of rounds, which is a function of Nk and Nb (which is fixed). For this standard, Nr = 10, 12, or 14.

Rcon[ ] The round constant word array.

RotWord() Function used in the Key Expansion routine that takes a four-byte word and performs a cyclic permutation.

ShiftRows() Transformation in the Cipher that processes the State by cyclically shifting the last three rows of the State by different offsets.

SubBytes() Transformation in the Cipher that processes the State using a nonlinear byte substitution table (S-box) that operates on each of the State bytes independently.

SubWord() Function used in the Key Expansion routine that takes a four-byte input word and applies an S-box to each of the four bytes to produce an output word.

XOR Exclusive-OR operation.

$\oplus$ Exclusive-OR operation.

$\otimes$ Multiplication of two polynomials (each with degree < 4) modulo $x^4 + 1$.

• Finite field multiplication.

A.2 Notation and Conventions

A.2.1 Inputs and Outputs

The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits** (digits with values of 0 or 1). These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits** . Other input, output and Cipher Key lengths are not permitted by this standard.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number i attached to a bit is known as its index and will be in one of the ranges

$0 \leq i < 128$, $0 \leq i < 192$ or $0 \leq i < 256$ depending on the block length and key length (specified above).

A.2.2 Bytes

The basic unit for processing in the AES algorithm is a **byte** , a sequence of eight bits treated as a single entity. The input, output and Cipher Key bit sequences described in Sec. A.2.1 are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits to form arrays of bytes (see Sec. A.2.3). For an input, output or Cipher Key denoted by a, the bytes in the resulting array will be referenced using one of the two forms, a_n or a[n], where n will be in one of the following ranges:

Key length = 128 bits , $0 \leq n < 16$; Block length = 128 bits , $0 \leq n < 16$;

Key length = 192 bits , $0 \leq n < 24$;

Key length = 256 bits , $0 \leq n < 32$;

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order $\{ b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0 \}$. These bytes are interpreted as finite field elements using a polynomial representation:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0 = \sum_{i=0}^{7} b_i x^i. \qquad \text{(A.1)}$$

For example, *{01100011}* identifies the specific finite field element x^6 + x^5 + x + 1.

It is also convenient to denote byte values using hexadecimal notation with each of two groups of four bits being denoted by a single character as in Table A.1 .

Hence the element $\{01100011\}$ can be represented as $\{63\}$, where the character denoting the four-bit group containing the higher numbered bits is again to the left.

Some finite field operations involve one additional bit (b_8) to the left of an 8-bit byte. Where this extra bit is present, it will appear as '$\{01\}$' immediately preceding the 8-bit byte; for example, a 9-bit sequence will be presented as $\{01\}\{1b\}$.

Big Pattern	Character	Big Pattern	Character
0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f

TABLE A.1: Hexadecimal representation of bit patterns

A.2.3 Array of Bytes

Arrays of bytes will be represented in the following form:

$$a_0\ a_1\ a_2\ ...\ a_{15}$$

The bytes and the bit ordering within bytes are derived from the 128-bit input sequence

$$input_0\ input_1\ input_2\ ...\ input_{126}\ input_{127}$$

as follows:

$$\begin{gathered} a_0 = \{input_0, input_1, ..., input_7\} \\ a_1 = \{input_8, input_9, ..., input_{15}\} \\ ... \\ a_{15} = \{input_{120}, input_{121}, ..., input_{127}\} \end{gathered}$$

The pattern can be extended to longer sequences (i.e., for 192- and 256-bit keys), so that, in general,

$$a_n = \{input_{8n}, input_{8n+1}, ..., input_{8n+7}\} \tag{A.2}$$

Taking Sections A.1.3.2 and A.1.3.2 together, Table A.2 shows how bits within each byte are numbered.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Byte number	0								1								...
Bit numbers in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	...

TABLE A.2: Indices for Bytes and Bits.

A.2.4 The State

Internally, the AES algorithm's operations are performed on a two - dimensional array of bytes called the **State**. The State consists of four rows of bytes, each containing **Nb** bytes, where **Nb** is the block length divided by 32. In the State array denoted by the symbol s, each individual byte has two indices, with its row number r in the range $0 \leq c < 4$ and its column number c in the range $0 \leq c < Nb$. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or s[r,c]. For this standard, **Nb** = 4, i.e., $0 \leq c < 4$.

At the start of the Cipher and Inverse Cipher described in Sec. 5, the input – the array of bytes $in_0, in_1, ...in_{15}$ – is copied into the State array as illustrated in Fig. 3. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes $out_0, out_1, ...out_{15}$.

input bytes					*State array*					*output bytes*			
in_0	in_4	in_8	in_{12}	$\Rightarrow$	$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$	$\Rightarrow$	out_0	out_4	out_8	out_{12}
in_1	in_5	in_9	in_{13}		$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$		out_1	out_5	out_9	out_{13}
in_2	in_6	in_{10}	in_{14}		$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$		out_2	out_6	out_{10}	out_{14}
in_3	in_7	in_{11}	in_{15}		$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$		out_3	out_7	out_{11}	out_{15}

TABLE A.3: State array input and output

Hence, at the beginning of the Cipher or Inverse Cipher, the input array, in, is copied to the State array according to the scheme:

$$s[r,c] = in[r+4c] \quad \text{for} \quad 0 \leq r < 4 \quad \text{and} \quad 0 \leq c < Nb, \tag{A.3}$$

and at the end of the Cipher and Inverse Cipher, the State is copied to the output array out as follows:

$$out[r+4c] = s[r,c] \quad \text{for} \quad 0 \leq r < 4 \quad \text{and} \quad 0 \leq c < Nb, \tag{A.4}$$

A.2.5 The State as an Array of Columns

The four bytes in each column of the State array form 32-bit words, where the row number r provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), w0...w3, where the column number c provides an index into this

array. Hence, for the example in Fig. 3, the State can be considered as an array of four words, as follows:

$$\begin{aligned} w_0 &= s_{0,0}\, s_{1,0}\, s_{2,0}\, s_{3,0} & \qquad w_2 &= s_{0,2}\, s_{1,2}\, s_{2,2}\, s_{3,2} \\ w_1 &= s_{0,1}\, s_{1,1}\, s_{2,1}\, s_{3,1} & \qquad w_3 &= s_{0,3}\, s_{1,3}\, s_{2,3}\, s_{3,3} \end{aligned} \tag{A.5}$$

A.3 Mathematical Preliminaries

All bytes in the AES algorithm are interpreted as finite field elements using the notation introduced in Sec. 3.2. Finite field elements can be added and multiplied, but these operations are different from those used for numbers. The following subsections introduce the basic mathematical concepts needed for Sec. 5.

A.3.1 Addition

The addition of two elements in a finite field is achieved by "adding" the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the XOR operation (denoted by $\oplus$) - i.e., modulo 2 - so that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, and $0 \oplus 0 = 0$. Consequently, subtraction of polynomials is identical to addition of polynomials.

Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7\, a_6\, a_5\, a_4\, a_3\, a_2\, a_1\, a_0\}$ and $\{b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0\}$, the sum is $\{c_7\, c_6\, c_5\, c_4\, c_3\, c_2\, c_1\, c_0\}$, where each $c_i = a_i \oplus b_i$ (i.e., $c_7 = a_7 \oplus b_7$, $c_6 = a_6 \oplus b_6$, ... $c_0 = a_0 \oplus b_0$).

For example, the following expressions are equivalent to one another:

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$ (polynomial notation)

$\{01010111\} \oplus \{10000011\} = \{11010100\}$ (binary notation)

$\{57\} \oplus \{83\} = \{d4\}$ (hexadecimal notation)

A.3.2 Multiplication

In the polynomial representation, multiplication in GF(2^8) (denoted by $\bullet$) corresponds with the multiplication of polynomials modulo an **irreducible polynomial** of degree 8. A polynomial is irreducible if its only divisors are

one and itself. **For the AES algorithm, this irreducible polynomial is**

$$m(x) = x^8 + x^4 + x^3 + x + 1, \tag{A.6}$$

or $\{01\}\{1b\}$ in hexadecimal notation.

For example, $\{57\} \bullet \{83\} = \{c1\}$, because

$$\begin{aligned}(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = \quad & x^{13} + x^{11} + x^9 + x^8 + x^7 + \\ & x^7 + x^5 + x^3 + x^2 + x + \\ & x^6 + x^4 + x^2 + x + 1 \\ = \quad & x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + \\ & x^4 + x^3 + 1\end{aligned}$$

and

$$\begin{aligned}x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \quad & modulo \quad (x^8 + x^4 + x^3 + x + 1) \\ = \quad & x^7 + x^6 + 1.\end{aligned}$$

The modular reduction by $m(x)$ ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication.

The multiplication defined above is associative, and the element $\{01\}$ is the multiplicative identity. For any non-zero binary polynomial $b(x)$ of degree less than 8, the multiplicative inverse of $b(x)$, denoted $b^{-1}(x)$, can be found as follows: the extended Euclidean algorithm [7] is used to compute polynomials $a(x)$ and $c(x)$ such that

$$b(x)a(x) + m(x)c(x) = 1. \tag{A.7}$$

Hence, $a(x) \bullet b(x) \bmod m(x) = 1$, which means

$$b^{-1}(x) = a(x) mod m(x). \tag{A.8}$$

Moreover, for any $a(x)$, $b(x)$ and $c(x)$ in the field, it holds that

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x).$$

It follows that the set of 256 possible byte values, with XOR used as addition and the multiplication defined as above, has the structure of the finite field GF(2^8).

A.3.2.1 Multiplication by x

Multiplying the binary polynomial defined in equation A.1 with the polynomial x results in

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x. \tag{A.9}$$

The result $x \bullet b(x)$ is obtained by reducing the above result modulo $m(x)$, as defined in equation A.6. If $b_7 = 0$, the result is already in reduced form. If $b_7 = 1$, the reduction is accomplished by subtracting (i.e., XORing) the polynomial $m(x)$. It follows that multiplication by x (i.e., $\{00000010\}$ or $\{02\}$) can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with $1b$. This operation on bytes is denoted by $xtime()$. Multiplication by higher powers of x can be implemented by repeated application of $xtime()$. By adding intermediate results, multiplication by any constant can be implemented.

For example,$\{57\} \bullet \{13\} = \{fe\}$ because

$$\begin{aligned}
\{57\} \bullet \{02\} &= xtime(\{57\}) = \{ae\} \\
\{57\} \bullet \{04\} &= xtime(\{ae\}) = \{47\} \\
\{57\} \bullet \{08\} &= xtime(\{47\}) = \{8e\} \\
\{57\} \bullet \{10\} &= xtime(\{8e\}) = \{07\},
\end{aligned}$$

thus,

$$\begin{aligned}
\{57\} \bullet \{02\} &= \{57\} \bullet (\{01\} \oplus \{02\} \oplus \{10\}) \\
&= \{57\} \oplus \{ae\} \oplus \{07\} \\
&= \{fe\}.
\end{aligned}$$

A.3.3 Polynomials with Coefficients in GF(2^8)

Four-term polynomials can be defined - with coefficients that are finite field elements - as:

$$a(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0 \tag{A.10}$$

which will be denoted as a word in the form $[a_0, a_1 0, a_2, a_3]$. Note that the polynomials in this section behave somewhat differently than the polynomials used in the definition of finite field elements, even though both types of polynomials use the same indeterminate, x. The coefficients in this section are themselves finite field elements, i.e., bytes, instead of bits; also, the multiplication of four-term polynomials uses a different reduction polynomial, defined below.The distinction should always be clear from the context.

To illustrate the addition and multiplication operations, let

$$b(x) = b_3x^3 + b_2x^2 + b_1x^1 + b_0 \tag{A.11}$$

define a second four-term polynomial. Addition is performed by adding the finite field coefficients of like powers of x. This addition corresponds to an XOR operation between the corresponding bytes in each of the words – in other words, the XOR of the complete word values.

Thus, using the equations of A.10 and A.11,

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x^1 + (a_0 \oplus b_0) \tag{A.12}$$

Multiplication is achieved in two steps. In the first step, the polynomial product $c(x) = a(x) \bullet b(x)$ is algebraically expanded, and like powers are collected to give

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x^1 + c_0 \tag{A.13}$$

where

$$\begin{aligned}
c_0 &= a_0 \bullet b_0 & c_4 &= a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \\
c_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_2 & c_5 &= a_1 \bullet b_0 \oplus a_0 \bullet b_2 \\
c_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 & c_6 &= a_3 \bullet b_3 \\
c_3 &= a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3
\end{aligned} \tag{A.14}$$

The result, $c(x)$, does not represent a four-byte word. Therefore, the second step of the multiplication is to reduce $c(x)$ modulo a polynomial of degree 4; the result can be reduced to a polynomial of degree less than 4. **For the AES algorithm, this is accomplished with the polynomial** $x^4 + 1$, so that

$$x^i \; mod \; (x^4 + 1) = x^{i \; mod \; 4} \tag{A.15}$$

The modular product of $a(x)$ and $b(x)$, denoted by $a(x) \oplus b(x)$, is given by the four-term polynomial $d(x)$, defined as follows:

$$d(x) = d_3x^3 + d_2x^2 + d_1x^1 + d_0 \tag{A.16}$$

with

$$\begin{aligned} d_0 &= (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3) \\ d_1 &= (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3) \\ d_2 &= (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3) \\ d_3 &= (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3) \end{aligned} \tag{A.17}$$

When $a(x)$ is a fixed polynomial, the operation defined in equation A.16 can be written in matrix form as:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{A.18}$$

Because $x^4 + 1$ is not an irreducible polynomial over GF(2^8), multiplication by a fixed four-term polynomial is not necessarily invertible. However, the AES algorithm specifies a fixed four-term polynomial that does have an inverse (see Sec. 5.1.3 and Sec. 5.3.3):

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x^1 + \{02\} \tag{A.19}$$

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x^1 + \{0e\} \tag{A.20}$$

Another polynomial used in the AES algorithm (see the RotWord()function in Sec. 5.2) has $a_0 = a_1 = a_2 = 00$ and $a_3 = 01$, which is the polynomial x^3. Inspection of equation (4.13) above will show that its effect is to form the output word by rotating bytes in the input word. This means that $[b_0, b_1, b_2, b_3]$ is transformed into $[b_1, b_2, b_3, b_0]$.

References

[1] URL: http://www.dredbox.eu/.

[2] Razvan Nane et al. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.

[3] *Wikipedia*. URL: https://www.wikipedia.com/.

[4] Muhammad Mushtaq et al. "A Survey on the Cryptographic Encryption Algorithms". In: *International Journal of Advanced Computer Science and Applications* 8 (Nov. 2017), pp. 333–344.

[5] Enas Elgeldawi, Maha Mahrous, and Awny Sayed. "A Comparative Analysis of Symmetric Algorithms in Cloud Computing: A Survey". In: *International Journal of Computer Applications* 182 (Apr. 2019), pp. 7–16. DOI: 10.5120/ijca2019918726.

[6] Soufiane Oukili and Seddik Bri. "FPGA implementation of Data Encryption Standard using time variable permutations". In: (2015), pp. 126–129. DOI: 10.1109/ICM.2015.7438004.

[7] C. Patterson. "High performance DES encryption in Virtex/sup TM/ FPGAs using JBits/sup TM/". In: (2000), pp. 113–121. DOI: 10.1109/FPGA.2000.903398.

[8] Vikram Pasham and Steve Trimberger. "High-speed DES and triple DES encryptor/decryptor". In: *Xilinx Application Notes* (Jan. 2001).

[9] Paris Kitsos et al. "64-bit Block ciphers: Hardware implementations and comparison analysis". In: *Computers & Electrical Engineering* 30 (Nov. 2004), pp. 593–604. DOI: 10.1016/j.compeleceng.2004.11.001.

[10] E. J. Swankoski et al. "A parallel architecture for secure FPGA symmetric encryption". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004, pp. 132–. DOI: 10.1109/IPDPS.2004.1303101.

[11] Umer Farooq and M. Faisal Aslam. "Comparative analysis of different AES implementation techniques for efficient resource usage and better performance of an FPGA". In: *Journal of King Saud University - Computer*

and Information Sciences 29.3 (2017), pp. 295 –302. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2016.01.004. URL: http://www.sciencedirect.com/science/article/pii/S1319157816300143.

[12] Reza Rezaeian Farashahi, Bahram Rashidi, and Sayed Masoud Sayedi. "FPGA based fast and high-throughput 2-slow retiming 128-bit AES encryption algorithm". In: *Microelectronics Journal* 45.8 (2014), pp. 1014 –1025. ISSN: 0026-2692. DOI: https://doi.org/10.1016/j.mejo.2014.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0026269214001505.

[13] S. S. H. Shah and G. Raja. "FPGA implementation of chaotic based AES image encryption algorithm". In: *2015 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*. 2015, pp. 574–577. DOI: 10.1109/ICSIPA.2015.7412256.

[14] S. Oukili and S. Bri. "High speed efficient advanced encryption standard implementation". In: *2017 International Symposium on Networks, Computers and Communications (ISNCC)*. 2017, pp. 1–4. DOI: 10.1109/ISNCC.2017.8071975.

[15] Soufiane Oukili and Seddik Bri. "High Throughput Parallel Implementation of Blowfish Algorithm". In: *Applied Mathematics & Information Sciences* 10 (Nov. 2016), pp. 2087–2092. DOI: 10.18576/amis/100611.

[16] S. B. Nalawade and D. H. Gawali. "Design and implementation of blowfish algorithm using reconfigurable platform". In: *2017 International Conference on Recent Innovations in Signal processing and Embedded Systems (RISE)*. 2017, pp. 479–484. DOI: 10.1109/RISE.2017.8378204.

[17] J. Sugier. "Implementation of symmetric block ciphers in popular-grade FPGA devices". In: vol. 3. 2. 2012, pp. 179–187.

[18] Jesús Lázaro et al. "High Throughput Serpent Encryption Implementation". In: Aug. 2004, pp. 996–1000. ISBN: 978-3-540-22989-6. DOI: 10.1007/978-3-540-30117-2_114.

[19] B. Najafi et al. "High speed implementation of Serpent algorithm". In: *Proceedings. The 16th International Conference on Microelectronics, 2004. ICM 2004.* 2004, pp. 718–721. DOI: 10.1109/ICM.2004.1434767.

[20] Kris Gaj and Pawel Chodowiec. "Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard Using Field Programmable Gate Arrays". In: vol. 2020. May 2001. ISBN: 978-3-540-41898-6. DOI: 10.1007/3-540-45353-9_8.

[21] D. Smekal, J. Hajny, and Z. Martinasek. "Hardware-Accelerated Twofish Core for FPGA". In: *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*. 2018, pp. 1–5. DOI: 10.1109/TSP.2018.8441386.

[22] Jean-Luc Beuchat. "FPGA Implementations of the RC6 Block Cipher". In: *Field Programmable Logic and Application*. Ed. by Peter Y. K. Cheung and George A. Constantinides. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 101–110. ISBN: 978-3-540-45234-8.

[23] Dimitris Theodoropoulos, Alexandros Siskos, and Dionisis Pnevmatikatos. "CCproc: A Custom VLIW Cryptography Co-processor for Symmetric-Key Ciphers". In: *Reconfigurable Computing: Architectures, Tools and Applications*. Ed. by Jürgen Becker et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 318–323. ISBN: 978-3-642-00641-8.

[24] J. L. Beuchat. "Modular multiplication for FPGA implementation of the IDEA block cipher". In: *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. ASAP 2003*. 2003, pp. 412–422. DOI: 10.1109/ASAP.2003.1212864.

[25] M. A. Hussain and R. Badar. "FPGA Based Implementation Scenarios of TEA Block Cipher". In: *2015 13th International Conference on Frontiers of Information Technology (FIT)*. 2015, pp. 283–286. DOI: 10.1109/FIT.2015.56.

[26] Jens-Peter Kaps. "Chai-Tea, Cryptographic Hardware Implementations of xTEA". In: *Progress in Cryptology - INDOCRYPT 2008*. Ed. by Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 363–375. ISBN: 978-3-540-89754-5.

[27] P. Kitsos, M. D. Galanis, and O. Koufopavlou. "A RAM-based FPGA implementation of the 64-bit MISTY1 block cipher". In: *2005 IEEE International Symposium on Circuits and Systems*. 2005, 4641–4644 Vol. 5. DOI: 10.1109/ISCAS.2005.1465667.

[28] G. Rouvroy et al. "Efficient FPGA implementation of block cipher MISTY1". In: *Proceedings International Parallel and Distributed Processing Symposium*. 2003, 7 pp.–. DOI: 10.1109/IPDPS.2003.1213343.

[29] François-Xavier Standaert et al. "Efficient FPGA Implementations of Block Ciphers KHAZAD and MISTY1". In: (Nov. 2002).

[30] Z. Čiča. "Pipelined implementation of Camellia encryption algorithm". In: *2016 24th Telecommunications Forum (TELFOR)*. 2016, pp. 1–4. DOI: 10.1109/TELFOR.2016.7818785.

[31] S.-W Lee, S.-J Moon, and J.-N Kim. "High-speed hardware architectures for ARIA with composite field arithmetic and area-throughput trade-offs". In: 30 (Oct. 2008), pp. 707–717.

[32] Federal Information Processing Standards Publication 197. "Announcing the ADVANCED ENCRYPTION STANDARD (AES)". In: (November 26, 2001). DOI: doi:10.6028/NIST.FIPS.197.

[33] Ronald L. Rivest et al. "The RC 6 TM Block Cipher". In: 1998. URL: http://people.csail.mit.edu/rivest/Rc6.pdf.

[34] Bruce Schneier. "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)". In: *Fast Software Encryption Workshop*. 1993. URL: https://www.schneier.com/academic/archives/1994/09/description_of_a_new.html.

[35] *Vivado HLS 2017.1*. URL: https://docs.xilinx.com/v/u/2017.1-English/ug902-vivado-high-level-synthesis.

[36] *The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)*. URL: https://csrc.nist.gov/Presentations/2002/The-Advanced-Encryption-Standard-Algorithm-Validat.

[37] URL: https://github.com/kokke/tiny-AES-c.