

PROCESSOR CUSTOMIZATION FOR LIGHTWEIGHT CRYPTOGRAPHY ON IOT DEVICES

A Thesis

by

Abdullah Varıcı

Submitted to the

Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the

Department of Electrical and Electronics Engineering

Özyeğin University

August 2020

Copyright © 2020 by Abdullah Varıcı

PROCESSOR CUSTOMIZATION FOR LIGHTWEIGHT CRYPTOGRAPHY ON IOT DEVICES

Advisor: Prof. Hasan Fatih Uğurdağ

Co-Advisor: Prof. Sezer Gören Uğurdağ

Approved by:

Prof. Hasan Fatih Uğurdağ, Advisor,
Department of Electrical and
Electronics Engineering
Özyeğin University

Assoc. Prof. Hasan Sözer,
Department of Computer Science
Özyeğin University

Asst. Prof. Onur Demir,
Department of Computer Engineering
Yeditepe University

Date Approved: 20 August 2020



To the Gorgeous People in my Life

ABSTRACT

This thesis was motivated by a request from a hardware security group we collaborate with at North Carolina State University, which in turn works with the National Institute of Standards and Technology (NIST) in the USA. NIST is in the process of evaluating many candidate lightweight cryptographic algorithms for Internet of Things (IoT) devices. The criteria are achieving one or more of reduced design time, reduced hardware personnel, reduced area, and reduced power/energy consumption. Such goals are applicable to other IoT applications as well. There are multiple design approaches that are possible such as pure Register Transfer Level (RTL) design (i.e., hardwired approach), High Level Synthesis (HLS), standard processor, and customized processor. We are proponents of customized processors. A complete IoT product needs a processor anyway. We used the renowned PRESENT cipher as a typical test case for lightweight cryptography for IoT. Through an implementation of PRESENT in a complete usecase we show that customized processor approach is able to achieve a smaller area even compared to pure RTL approach. Another contribution of this thesis to the literature is the implementation of PRESENT in various ways, namely, pure RTL, HLS, standard processor, customized processor approach, and comparing these with the results in the literature. Yet another contribution is that this work serves as a usecase for our work on processor customization methodology development (MC2SC). Some of our PRESENT designs benefited from the MC2SC project and, in return, allowed us to improve the tools that resulted from the MC2SC project.

ÖZETÇE

Bu tez, Kuzey Carolina Eyalet Üniversitesi'nde işbirliği yaptığımız ve ABD'deki Ulusal Standartlar ve Teknoloji Enstitüsü (NIST) ile birlikte çalışan bir donanım güvenlik grubundan gelen talep üzerine motive olup oluşturulmaya başlandı. NIST, Nesnelerin İnterneti (IoT) cihazları için birçok hafif kriptografik algoritma adayını değerlendirme sürecindedir. Daha kısa tasarım süresi, daha az sayıda tasarımcı, daha az alan ve daha az güç/enerji tüketimi kriterler arasındadır. Bu tür hedefler, birçok IoT uygulaması için geçerlidir. Yazmaç Transfer Seviyesi (RTL) tasarımı, Yüksek Seviye Sentez (HLS), standart işlemci ve özelleştirilmiş işlemci gibi mümkün olan birden fazla tasarım yaklaşımı mevcuttur. Biz özelleştirilmiş işlemcileri savunmaktayız. Tamamlanmış bir IoT ürünü zaten bir işlemciye ihtiyaç duymaktadır. IoT uygulamalarındaki hafif kriptografiye bir test örneği olması için popüler PRESENT blok şifreleme algoritmasını kullandık. PRESENT algoritmasının IoT cihazlar üzerinde gerçeğe yakın bir kullanım senaryosunda özelleştirilmiş işlemci yaklaşımının, saf RTL yaklaşımına kıyasla daha küçük bir alan elde edebildiğini gösteriyoruz. Bu tezin literatüre bir başka katkısı da PRESENT'in çeşitli şekillerde, yani saf RTL, HLS, standart işlemci, özelleştirilmiş işlemci ile gerçekleştirilmesi ve bunların literatürdeki sonuçlarla karşılaştırılmasıdır. Bu çalışmanın yine bir başka katkısı da işlemci özelleştirme metodolojisi geliştirme (MC2SC) projemiz için bir kullanım senaryosu oluşturmaktır. PRESENT tasarımlarımızdan bazıları MC2SC projesinden yararlandı ve karşılığında MC2SC projesinde oluşturulan araçları iyileştirmemize olanak sağladı.

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Prof. H. Fatih Uğurdağ for accepting me as his student. His courses and projects are very educational, and I have learned a lot from him about the professional life out there. He was very friendly and helpful in supervising, guiding, and mentoring me. He set realistic goals/deadlines and trusted me. I am grateful for his support and patience, which enabled me to complete this thesis. I would also like to thank my co-advisor Prof. Sezer Gören Uğurdağ and the members of the thesis defense committee, Assoc. Prof. Hasan Sözer and Asst. Prof. Onur Demir for their valuable time.

I also want to express my thanks to my employer, Procenne, for making me a part of a great team and great projects as well as their interest in my continued education. My thanks also go to the members of nEMESysLab. They made my time at the university enjoyable, thanks to the friendly atmosphere there.

I would like to extend my gratitude to TÜBİTAK for supporting me through their 2211 Scholarship Program. Please also note that this thesis was done in conjunction with a TÜBİTAK 1001 project (no: 117E090). The thesis used some tools developed by the TÜBİTAK project while providing a usecase for it. Prof. Sezer Gören Uğurdağ is the Principal Investigator (PI) of the project. Initially, Asst. Prof. T. Barış Aktemur was the co-PI, and later, Prof. H. Fatih Uğurdağ replaced him as the co-PI.

Finally, my special thanks go to my family and friends for their unconditional support, friendship, love, and encouragement throughout my life.

TABLE OF CONTENTS

ABSTRACT	iv
ÖZETÇE	v
ACKNOWLEDGMENTS	vi
LIST OF TABLES.....	ix
LIST OF FIGURES.....	x
I. INTRODUCTION	1
1.1 Cryptography	1
1.2 Internet of Things.....	4
1.3 Lightweight Cryptography.....	4
1.4 Motivation.....	6
1.5 Contributions of the Thesis.....	7
II. BACKGROUND.....	9
2.1 Integrated Circuits	9
2.2 Metrics	11
2.2.1 Area	11
2.2.2 Latency	11
2.2.3 Throughput.....	11
2.2.4 Power and Energy	12
2.2.5 Design Time and Design Team.....	12
III. PRESENT BLOCK CIPHER	13
3.1 addRoundKey.....	15
3.2 sBoxLayer.....	15
3.3 pLayer	16
3.4 keySchedule	16
IV. IMPLEMENTATIONS OF PRESENT BLOCK CIPHER.....	18

4.1	With Hand-Coded RTL	19
4.1.1	Design Details	19
4.1.2	Design Results.....	20
4.2	With Vivado HLS.....	20
4.2.1	Design Details	20
4.2.2	Design Results.....	21
4.3	With a Standard Processor.....	21
4.3.1	Design Details	23
4.3.2	Design Results.....	23
4.4	With VSCPU.....	24
4.4.1	Design Details	27
4.4.2	Design Results.....	27
4.5	With Customized VSCPU	27
4.5.1	Design Details (VSCPU-80).....	28
4.5.2	Design Results (VSCPU-80)	31
4.5.3	Optimized Design Details (VSCPU-16).....	31
4.5.4	Optimized Design Results (VSCPU-16)	33
4.6	Comparison of Our Implementations and Other Works	33
4.7	A Use Case Scenario for PRESENT in an IoT Application.....	37
4.7.1	Design Details	38
4.7.2	Design Results.....	39
v.	CONCLUSIONS AND FUTURE WORK	41
	APPENDIX A – Source Code for Design 1	44
	APPENDIX B – Source Code for Design 2	54
	BIBLIOGRAPHY	67
	VITA	69

LIST OF TABLES

1. Instruction word structure of VSCPU [10]	25
2. Instruction set of VSCPU [10]	26
3. Instruction set of VSCPU-80.....	29
4. Handwritten assembly code of PRESENT algorithm for VSCPU-80.....	30
5. Instruction set of VSCPU-16.....	32
6. Memory related information on processor based implementations.....	34
7. Comparison of PRESENT implementations on FPGA	34
8. Implementation results of Design 1 and Design 2.....	40

LIST OF FIGURES

1. The illustration of encryption and decryption operations [2].....	2
2. The scytale [4]	3
3. The IoT platform produces solutions to many daily life problems [6]	4
4. Internal structure of FPGAs [13]	10
5. PRESENT block cipher encryption algorithm	14
6. Substitution box layer of PRESENT	15
7. Permutation layer of PRESENT	16
8. Key schedule algorithm of PRESENT.....	17
9. Block diagram of PicoBlaze [22]	21
10. Example C program and assembly code generated by C compiler [10].....	25
11. Design flow of the customized VSCPU.....	30

CHAPTER I

INTRODUCTION

People became interested in hiding or protecting information since ancient times. The development of smart electronic devices and information technology in the last decades has also increased the need for information security. In this chapter, some information about the cryptography, internet of the things, and the lightweight cryptography will be provided.

1.1 Cryptography

Secure and secret communication has been important since ancient times of human history. People have tried to hide the meaning of the message and at the same time, they tried to retrieve secret information in other people's messages.

Cryptology is the science that studies the communication and storage of data in a secure and generally secret form [1]. Cryptography and cryptanalysis are two main parts of the cryptology. Cryptography includes algorithms and protocols with mathematical techniques to provide information security, such as data integrity, confidentiality, and authentication. Contrary to this, the aim of the cryptanalysis is to find weaknesses in cryptographic algorithms and retrieving secret information using mathematical techniques.

In cryptography, the plaintext is the source of the information or message to be secured. The key is referred to as the secret information that is known only by

legitimate users. Encryption is the process of transforming the plaintext by using the key to the ciphertext, which is the secret and usually meaningless form of the message. The inverse operation of the encryption is known as decryption. With decryption, the original form of the information (plaintext) is recovered from ciphertext using the key. The relation between these terms is depicted in Figure 1.

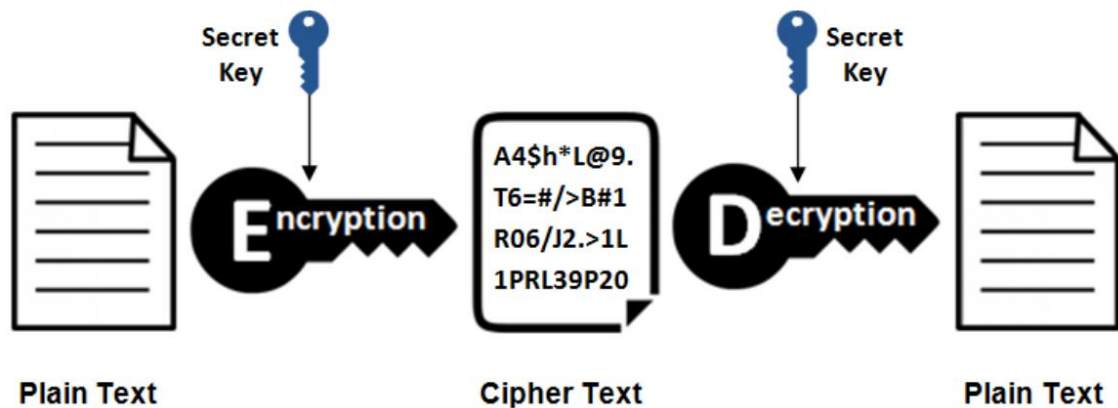


Figure 1: The illustration of encryption and decryption operations [2]

There are two types of cryptography: symmetric cryptography and asymmetric cryptography. In symmetric cryptographic algorithms, the same secret key is used for encryption and decryption. However, in asymmetric cryptosystems, different but mathematically related keys are used for encryption and decryption. Both types of cryptography have advantages and disadvantages and they may be used together or separately in so many cryptographic protocols. Furthermore, symmetric cryptography can be divided into two main branches: block ciphers and stream ciphers. In stream ciphers, plaintexts are encrypted one bit at each time. However, in block ciphers, the plaintext is divided into blocks with fixed length and they are encrypted one block of plaintext at each time. Advanced Encryption Algorithm (AES) is one of the most popular cryptographic algorithms in the world. It is a symmetric block cipher algorithm with a block length of 128 bit and a key length of 128, 192, or 256 bit.

In ancient times, Greeks used a tool named scytale, which is a cylinder for stripping a parchment around it to write and recover secret messages during military campaigns as in Figure 2 [3]. Later, Caesar cipher is used by Roman emperors for secret communication by replacing every letter in the message in another letter. Then, in the first half of the 20th century, the transition from classical cryptography to the modern cryptography is started. The Enigma machine is one of the famous military applications of modern cryptography. Both cryptography and cryptoanalysis continued to develop simultaneously after World War II in parallel to the developments in electronic science. In earlier years of modern cryptography, applications were mostly military purposes. However, after the development of personal computers and with the spread of the Internet, cryptography has started to play a critical role in our lives, as it is at the heart of information security and communication. Nowadays, it is used in so many domains such as banking, commerce, and industry. Without any doubt, the importance and the usage of cryptography will be increased with the expansion of the Internet of Things.



Figure 2: The scytale [4]

1.2 Internet of Things

The Internet of Things (IoT) is a platform where smart home appliances, various sensors, wearable, and many other devices communicate with each other over the Internet. The smart devices surrounding us gather information from the environment and they share the information by using processors and communication units with other smart devices over IoT platform to produce solutions to daily life problems. These smart devices are becoming more and more widely used in various fields such as financial, health, automotive, education, and urbanism. Such a situation makes information security in these devices covering many areas of our lives an important issue [5].

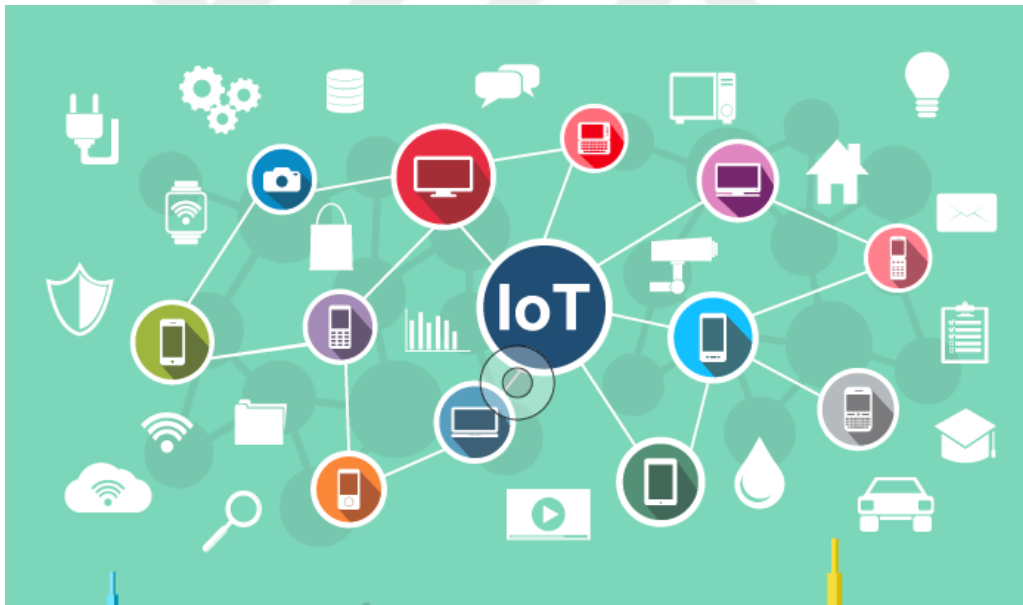


Figure 3: The IoT platform produces solutions to many daily life problems [6]

1.3 Lightweight Cryptography

IoT devices have characteristic customizations for specific applications. They must meet some constraints when operating autonomously to satisfy desired features such as

the price or size of the final product. Therefore, cryptographic primitives for the target security should also meet some constraints such as power/energy consumption, silicon area, and latency for hardware implementations; power/energy consumption, execution time, and code size for software implementations. On the other hand, generally, just a few sensor data, counter, or identifier information are processed and communicated between IoT devices. Therefore, usually, there is no need for high performance or throughput in IoT devices.

The messages that are sent from a device to another device may be read and intercepted by unauthorized people from the outside world. To prevent unwanted access to confidential information, data encryption before transfer and storage is a typical method. Encryption algorithms such as Advanced Encryption Standard (AES) [7] and Data Encryption Standard (DES) [8] are widely used for this purpose. However, since these algorithms require a large number of hardware resources and high power consumption, it is not possible to use them in edge/IoT devices with limited resources. The Lightweight Cryptography field [9] has emerged as a solution to this problem by decreasing processing complexity and the area usage of the chips while providing a reasonable amount of security.

National Institute of Standards and Technology (NIST) in the USA selects cryptographic standards by organizing some contests. DES [8] and AES [7] are standardized by NIST in the past and today there is also a need for the selection of a standard lightweight cryptographic algorithm for IoT applications. Because of that, NIST organizes workshops on lightweight cryptography. As in the previous standards, the algorithms are evaluated in terms of security and hardware/software implementation. Area consumption, design complexity, and power/energy consumption

are important criteria for hardware/software implementation of the lightweight cryptographic algorithms.

1.4 Motivation

Today, the interconnectivity of everyday electronic devices is expanding more than ever before. The smart devices surrounding us gather information from the environment around them, and they process/share the information with other smart devices and/or the cloud to increase the quality of people's lives (thus the IoT platform). For instance, someone's fridge, heart monitor, bathroom scale, and shoes can work together to monitor his cardiovascular health.

The mass deployment of the IoT devices brings along with its harsh cost and size constraints on electronic technology. The cost and size constraints of IoT devices require energy, power, and area consumption of the integrated circuits to be minimum. Fortunately, usually, only a few data are processed and communicated from/to IoT devices. Therefore, high throughput is not a requirement in most cases.

Although IoT devices have many advantages in our lives, there are also many risks in terms of security. Because these devices may process and communicate some security sensitive information in the military, financial, automotive, and many other applications. Lightweight cryptography is the solution since it provides enough security solutions for IoT systems with constrained resources.

This work was motivated by a request from a hardware security group we collaborate with at North Carolina State University, which in turn works with the National Institute of Standards and Technology (NIST) in the USA. NIST is in the process of evaluating many candidate lightweight cryptographic algorithms for Internet

of Things (IoT) devices. The criteria are achieving one or more of reduced design time, reduced hardware personnel, reduced area, and reduced power/energy consumption. Such goals are applicable to other IoT applications as well.

It is very challenging to implement cryptographic primitives by satisfying a specified set of constraints that depend on use case scenarios, and this requires expertise in different domains such as cryptography, computer science, and electronics engineering. This work aims to accomplish designing energy and area efficient implementation of a lightweight cryptographic algorithm for embedded IoT systems.

1.5 Contributions of the Thesis

In modern IoT systems, a classic use case scenario may be described as follows. Initially, some pre-computation is applied to the data gathered by sensors. Then they are encrypted to ensure information security. Finally, they are shared with other smart devices using some communication protocols. Ensuring this functionality with minimum energy, power, and area consumption on an integrated circuit is a challenging task and the main goal of this work.

One of the most important components of modern embedded systems are the processors and the software enables the processors to perform the desired operations in order. The instruction sets of many commercially available processors are non-customizable. This leads to the fact that the instructions, which the applications sometimes do not use at all, consume unnecessary hardware resources. Besides, much-needed application specific instructions are not in the instruction set causing serious decreases in performance and increases in memory usage. To avoid these problems, there is a need for application specific hardware design, and changing the instruction set and/or customization of processors.

A complete IoT product needs a processor anyway for multiple tasks such as protocol handling and processing of sensor values other than cryptographic operations. Therefore, the implementations should be evaluated by taking into consideration this whole functionality and we suggest that customized processors are the best choices for this kind of functionality and the criteria for the IoT applications.

We used the renowned PRESENT cipher as a typical test case for lightweight cryptography for IoT. Through an implementation of PRESENT in a complete usecase we show that customized processor approach is able to achieve a smaller area even compared to pure RTL approach. Another contribution of this thesis to the literature is the implementation of PRESENT in various ways, namely, pure RTL, HLS, standard processor, customized processor approach, and comparing these with the results in the literature. Yet another contribution is that this work serves as a usecase for our work on processor customization methodology development (Multi-Core Microcontroller Synthesis from C - MC2SC). MC2SC methodology provides a design flow by generating application-specific microcontrollers from the C program. Some of our PRESENT designs benefited from the MC2SC project and, in return, allowed us to improve the tools that resulted from the MC2SC project.

CHAPTER II

BACKGROUND

In this chapter, some background information about integrated circuits and metrics will be provided.

2.1 Integrated Circuits

Devices with electronic circuits are everywhere in our lives and the technology for developing and creating them had significant improvements since the 1950s with the first transistors. First Integrated Circuit (IC) was developed by embedding a few transistors into silicon and in time, Very Large Scale Integration (VLSI) technology is introduced, which has combined thousands of transistors, CPU, ROM, RAM, and other glue logic into a single chip, with the help of the rapid advances in this domain. Usually, these integrated circuits are developed for a specific application and therefore they are called Application-Specific Integrated Circuits (ASICs). The logic function in ASIC cannot be changed after manufacturing since its digital circuitry is created by permanently connected gates in silicon.

Field Programmable Gate Array (FPGA) is an IC made up of thousands of Configurable Logic Blocks (CLBs) with programmable interconnections. So, an FPGA can be reprogrammed in the field after manufacturing and it can be utilized for various applications. The internal structure of an FPGA is depicted in Figure 4.

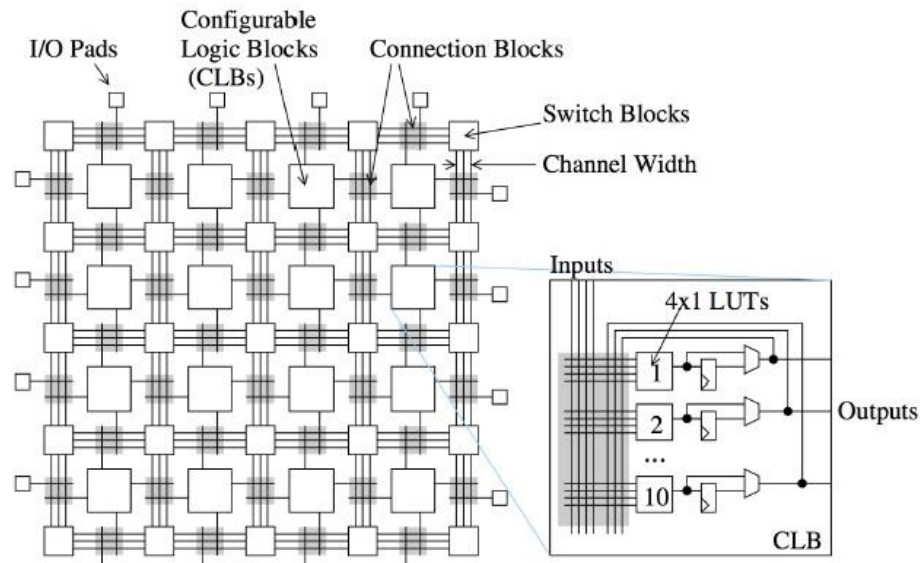


Figure 4: Internal structure of FPGAs [13]

In modern electronic technology, the structure of an integrated circuit is described through a hardware design flow with a hardware description language (VHDL or Verilog). The implementation of a digital circuit may be on a chip using FPGA or ASIC with similar processes but in the end, FPGA can be reprogrammed for different applications contrary to ASIC which can be used only for a specific application. However, ASICs are more energy efficient and they have higher performance than FPGAs.

Later, in the 1970s, Intel developed the first microprocessor. With this new technology, programmability is provided in ASIC although it is limited. Processors are one of the most important components of modern electronic systems. The instruction set of a processor determines the functionality and the capability of the processor. The software enables the processor to perform desired operations and application by running instructions in an order which are specified in the instruction set of the processor. After Intel developed the first microprocessor in the 1970s, programmability in ASIC is

provided although it is limited since the logic units and instruction set of the processor cannot be changed.

In this work, FPGAs are used as the target platform for implementations as they are used for prototyping most of the time.

2.2 Metrics

The following metrics are used as constraints for IoT applications and to assess the efficiency of an implementation.

2.2.1 Area

This metric depends on the standard cell library and the fabrication technology of the chips. Gate equivalent (GE) is used for comparison of chips with different technologies. However, in FPGAs slice, LUT and FF counts are used for this metric. This metric has a big impact on the power and energy consumption of the chips.

2.2.2 Latency

The amount of time required to perform an operation i.e. time between output and input for a specific operation. It can be either in clock cycles or in seconds.

2.2.3 Throughput

This metric is related to the performance of the implementation. The output rate of the implementation with respect to time. Bits per second (bps) is used for the expression.

2.2.4 Power and Energy

The electrical power and energy consumption of an IC. Power consumption is mostly in mW and this metric is very important for RFID applications. An implementation for a low cost passive smart device, such as RFID tags or contactless smart cards requires small area and power consumption, while the throughput is of secondary interest.

The energy consumption is equal to the total power consumption over a period. For battery-powered IoT devices, this metric is very important. The energy and power consumption of an IC is highly related to the area usage, manufacturing technology, latency of the operation, and clock frequency.

2.2.5 Design Time and Design Team

The design time is the total time for designing, implementing, and simulating an algorithm. It depends on the complexity of the algorithm, and the tools and methodologies for a specific implementation. The size of the design team is also related to the complexity of the algorithm, and the tools/methodologies.

CHAPTER III

PRESENT BLOCK CIPHER

Popular block ciphers are not suitable for constrained environments in IoT applications. The Lightweight Cryptography field has emerged as a solution to the area and energy consumption limitations of the IoT devices [9]. In 2007, Bogdanov et al. presented a new lightweight block cipher algorithm: PRESENT [11]. Researchers have published quite a few lightweight cryptographic algorithms for constrained IoT devices before PRESENT cipher. HIGHT was published by Hong et al in 2006 [14]. Lim and Korkishko published mCryton in 2005 [15]. Scalable Encryption Algorithm (SEA) was designed by Standaert et al. in 2006 [16]. Wheeler and Needham proposed TEA [17]. In 2004, ICEBERG was presented by Standaert et al. [18]. These were lightweight block ciphers for software and/or hardware implementations on IoT devices.

The aims when designing the PRESENT cipher were moderate security, efficient implementation, and simplicity. This algorithm was not developed for wide-spread use, some specific applications with the following characteristics were targeted [11]:

- Hardware implementation was targeted.
- 80-bit moderate security level is required. This is the primary consideration.
- Lower physical space ie. chip area was a secondary consideration.
- The third important metric for developing PRESENT was the peak and average power consumption.
- Timing requirement ie. latency for an operation was another consideration.

With the above considerations, the PRESENT cipher was developed as a 64-bit block cipher with an 80-bit key. An encryption-only PRESENT cipher implementation requires much fewer resources with help of the computation of encryption round keys on-the-fly, meaning every round key is generated from the previous round key before using in the encryption round. After all, PRESENT cipher has similar requirements to the previous lightweight block ciphers, but it consumes less chip area.

The PRESENT cipher algorithm is a lightweight symmetric-key block cipher algorithm. It has the Substitution Permutation Network (SPN) structure and has a 64-bit block length along with an 80-bit or 128-bit key length. The version with 80-bit keys is recommended and it provides enough security for IoT applications. For the applications that demand the most efficient use of space, only the encryption mode will be implemented.

As shown in Figure 5, the PRESENT block cipher encryption algorithm consists of 31 rounds and the last round of key addition for post-whitening. A round consists of the following 3 stages: add round key (addRoundKey), substitution box layer (sBoxLayer), and permutation layer (pLayer). The pseudo-code for the encryption routine of the cipher is depicted in Figure 5.

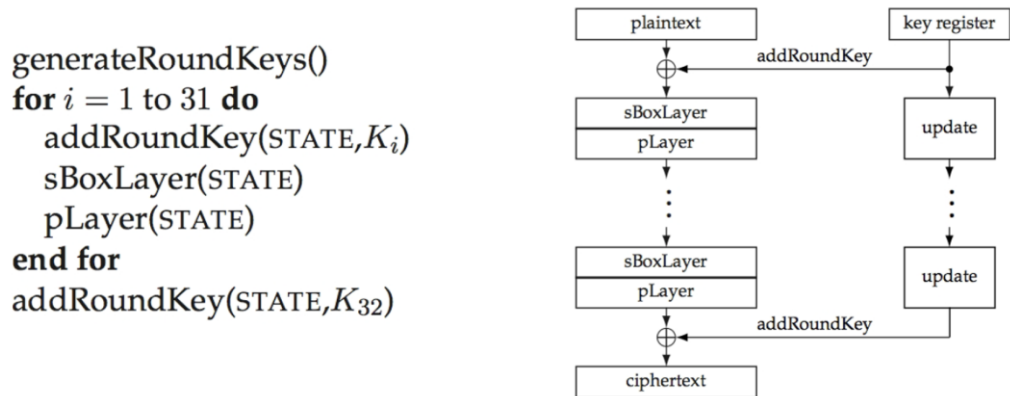


Figure 5: PRESENT block cipher encryption algorithm

At the start of the encryption algorithm, 64-bit plaintext is assigned to 64-bit state. At addRoundKey stage, 64-bit round input state is introduced to XOR operation with a round key K_i for i starting from 1 to 31. The main key is 80-bit or 128-bit length but the round keys are 64-bit length and they are generated by key schedule algorithm on-the-fly for encryption. In the next stage, at sBoxLayer, the state is divided 16 4-bit blocks and each block is updated with the same S-box transformation. A linear bitwise permutation is applied to the updated state at the pLayer stage. These round operations are repeated 31 times and after the final round, the round output state is introduced to XOR operation with the round key K_{32} . The result of this final operation is the ciphertext.

3.1 addRoundKey

Adding the round key is simply the XOR operation of the corresponding bits of the state data block and the round key.

3.2 sBoxLayer

The nonlinear substitution layer uses identical Substitution Box (SBox) that converts a 4-bit input to a 4-bit output. Figure 6 shows the input-output relationship of a single Sbox. Using identical Sboxes increases hardware implementation efficiency and this Sbox is selected by designers as it is resistant to differential and linear attacks [11].

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Figure 6: Substitution box layer of PRESENT

3.3 *pLayer*

In the permutation layer, the bit value in the index i of the state is moved to the index $P(i)$ for all indexes. Figure 7 provides the construction of the P function.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Figure 7: Permutation layer of PRESENT

3.4 *keySchedule*

PRESENT block cipher encryption algorithm takes keys of either 80-bit or 128-bit. In our work, we use the 80-bit version as it provides enough security for IoT applications. At each round, distinct round-keys are used. At first, the input key is stored in a register K . It is represented as $k_{79}k_{78} \dots k_0$. The 64-bit round key K_i at round i is the leftmost 64 bits of the current key register. At the start of each round, the round key is extracted from key register K and it is used in the `addRoundKey` stage. Then, the key register is updated as described in Figure 8.

1. $[k_{79}k_{78} \dots k_1k_0] = [k_{18}k_{17} \dots k_{20}k_{19}]$
2. $[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$
3. $[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus \text{round_counter}$

Figure 8: Key schedule algorithm of PRESENT

Initially, the key register is rotated by 61-bit positions to the left (or 19-bit positions to the right). Then, the leftmost 4 bits are passed through the Sbox, which is described in Figure 6. Finally, the round counter value i is subjected to XOR operation with the bits of $k_{19}k_{18}k_{17}k_{16}k_{15}$ indexes of the previous result. The most significant 64-bit of the value forms the round key.

CHAPTER IV

IMPLEMENTATIONS OF PRESENT BLOCK CIPHER

In the previous chapter, the PRESENT block cipher algorithm was introduced. In this chapter, the PRESENT algorithm is implemented with different design methodologies including hand-coded RTL, Vivado HLS, a standard processor (PicoBlaze), VerySimpleCPU, and customized VerySimpleCPU. The results are evaluated with the metrics described in chapter 2.2. The previous work is also reviewed, and their implementation results are compared with ours. A simple use case scenario example for a complete IoT application is also used for comparing some of the implementations.

Designing a custom chip is the first option that comes to mind when designing hardware with low resource and energy consumption. However, because of the high cost per product of chip design for small quantities of products, Field Programmable Gate Arrays (FPGA) are preferred. FPGAs are also used to ensure shorter design time and prototyping of ASICs. Therefore, FPGAs are used in this work for the target implementation. Xilinx's Vivado software with High Level Synthesis (HLS) feature is used for synthesis, high-level synthesis, simulation, and placement and routing. Xilinx XC7A100T-1CSG324 FPGA is selected as the target FPGA core. Only 80-bit key version of the PRESENT block cipher algorithm with encryption mode is used in this work since it provides enough security for IoT applications.

4.1 With Hand-Coded RTL

Before carrying out other implementations, we wanted to find out hardware resource consumption and typical performance of the PRESENT algorithm with hand-coded, i.e., pure RTL design on FPGA. For this purpose, we reviewed the RTL design shared on GitHub [19].

The most popular method when designing digital hardware i.e. integrated circuits that comes is Register Transfer Level (RTL) coding. In this way, the functionality of the digital hardware is described cycle and bit accurate level. This method is used since the 1980s by HDL-Synthesis tools. These tools transfer RTL design into a gate-level netlist either for ASIC or FPGA. In this work, Xilinx Vivado software is used for RTL coding, synthesis, implementation, and simulation for FPGAs.

4.1.1 Design Details

The entire design consists of 4 combinatorial logic modules for addRoundKey, sBoxLayer, pLayer, and keySchedule, and a simple finite state machine, which controls the whole operation [19]. With start signal, the first round begins by capturing plaintext and key inputs into the input state and key registers. Then, in every clock cycle key schedule and round operation is applied and the value of state and key registers are updated until the last round. After the final round, the PRESENT block cipher encryption operation is completed, and the value of the state register is given at the output (ciphertext).

4.1.2 Design Results

The RTL design is simulated, synthesized, and implemented on the selected FPGA without the need for optimization. The design consumed 213 LUTs and 213 FFs from FPGA resources after implementation. It also can work with a clock frequency up to 368 MHz, and it can compute a result in 32 clock cycles (latency).

4.2 With Vivado HLS

Using a High-Level Synthesis (HLS) software is another method for the implementation of the PRESENT block cipher algorithm, and it ensures a shorter design time.

HLS is a way of creating digital hardware designs by using software design coding syntaxes like C, C++. HLS enables digital hardware design automation to be increased with the transition from the traditional way of RTL coding to using classical software syntaxes in hardware design. In other words, HLS provides a higher level of abstraction when describing the functionality of a digital circuit. To make FPGA programming simpler and reduce the design time, Xilinx has developed the Vivado HLS. Thanks to this program, a C-based code can be transformed into an RTL design which allows synthesis and implementation on FPGAs [20].

4.2.1 Design Details

The C-based design of the PRESENT algorithm from FELICS (Fair Evaluation of Lightweight Cryptographic System) project [21] is used here. FELICS is a benchmarking framework designed for consistent and fair comparison and evaluation of different lightweight cryptographic algorithms' software implementations on embedded devices. The HLS based design is confirmed on the simulation that the design works

correctly. Afterward, synthesis and implementation of the obtained design from HLS were performed.

4.2.2 Design Results

The resulting design consumes 466 LUTs and 765 FFs, achieves a clock frequency of up to 125 MHz and takes 29124 cycles to complete an encryption operation on the same FPGA.

4.3 With a Standard Processor

In this work, Xilinx standard processor, PicoBlaze, is also used for the implementation of the PRESENT block cipher. With a processor, some logic resources may be re-used for the functions in the algorithm and programming would be much easier than coding RTL.

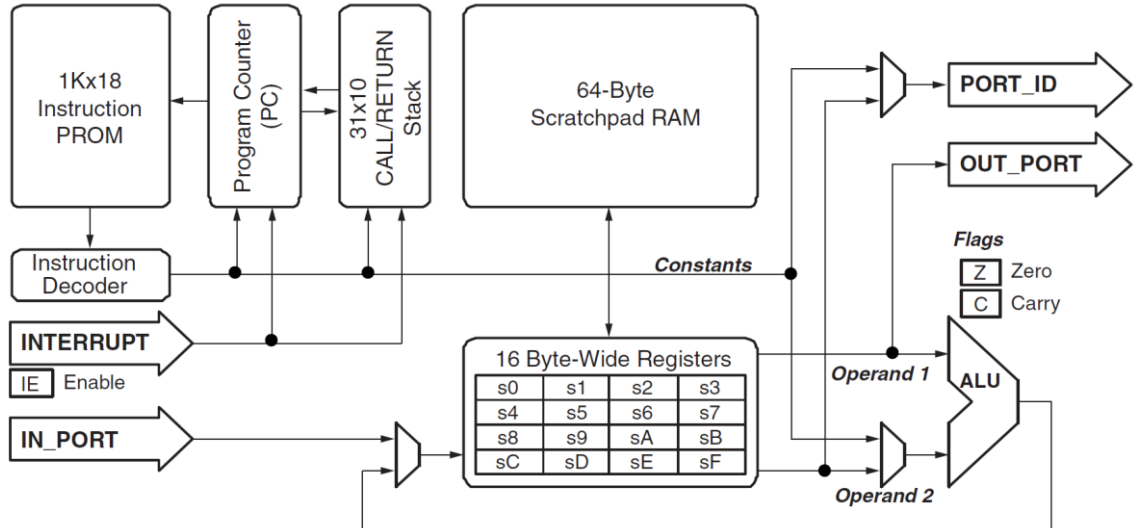


Figure 9: Block diagram of PicoBlaze [22]

PicoBlaze is an 8-bit processor with low area consumption and low performance for Xilinx's own FPGAs [22]. It is based on 8-bit Reduced Instruction Set Computing (RISC) architecture and this processor goes through the synthesis and implementation stages and works on the resources in Xilinx FPGAs. There is a total of 70 instructions in PicoBlaze and all instructions are executed in two clock cycles. The PicoBlaze can perform up to 100 million instructions per second (MIPS) depending on the target FPGA.

The block diagram of the PicoBlaze in Figure 9 shows the important functional blocks and I/O. PicoBlaze has the following features [22]:

- 16-byte general purpose data registers
- 64-byte internal scratchpad RAM
- 8-bit Arithmetic Logic Unit (ALU) with ZERO and CARRY flags including basic arithmetic operations like addition or subtraction, bitwise logic operations like AND, OR and XOR, arithmetic compare and bitwise test operations, shift and rotate operations
- I/O ports supporting data from/to 256 different addresses allowing to connect FPGA logic
- Interrupt support
- 1K instruction (program) memory which may be built with FPGA BRAM or registers
- CALL/RETURN stack with 31-location
- Assembler and instruction set simulator support

4.3.1 Design Details

The design files of the PicoBlaze are taken from the Xilinx website. In order to implement the PRESENT algorithm with PicoBlaze, the algorithm was written using Assembly language and PicoBlaze instruction set. Program memory for the PRESENT algorithm of the processor is created from this assembly code by the PicoBlaze assembler. Initially, the Assembly program consisted of 1805 instructions, and the design's program memory would consume a significant amount of resources. PicoBlaze design files with the generated program memory are simulated and expected results are seen at the end of the simulation.

To optimize the code, we reviewed it and reduced the number of assembly instructions to 509 by using some loops and some functions. Since FFs are used as the memory element in all implementations and to make the comparison under equal conditions, the memory elements of PicoBlaze are changed to FFs and LUTs from Block RAMs. Afterward, we performed synthesis and implementation using the PicoBlaze's source code and program memory file generated by the assembler.

4.3.2 Design Results

Eventually, the design working with a clock frequency up to 185 MHz consumed 292 LUTs and 2280 FFs and the latency for an encryption operation was 359943 clock cycles.

4.4 With VSCPU

VerySimpleCPU (VSCPU) is another processor that is also used for the PRESENT block cipher implementation in this work. VSCPU is selected since it is a customizable processor with a simple structure. In this section, it is used without any modifications.

VSCPU is a simple and customizable 32-bit soft processor that can be implemented on FPGAs, with an instruction set simulator, assembler, and C compiler [10]. Multi-Core Microcontroller Synthesis from C (or shortly MC2SC) methodology uses VSCPU processor, tools, and compilers to generate application-specific microcontrollers from C programs. VSCPU processor is also targeted to be used as teaching material for students learning computer architecture and organization. Students may review and customize the RTL code, instruction set simulator, assembler, and C compiler and have a better understanding of CPUs by practicing on this simple processor.

VSCPU has 16 instructions in its instruction set and supports unsigned integer arithmetic operations. The RTL codes of the processor are written in Verilog language and the structure can be changed completely as required. It has interrupt support and some peripherals like GPIO, SPI, UART are available. There are four registers used in the VSCPU:

- PC: Program Counter
- IW: Instruction Word
- R1: General Purpose Register
- R2: General Purpose Register

To complete VSCPU instructions takes 3 or 4 clock cycles depending on the instruction type. 32-bit instruction word format is shown in Table 1. Opcode and immediate bits define the type of operations. Operand A and B can refer to operand addresses, destination addresses, or value of the operand itself. In Table 2, the instruction set of VSCPU is shown with descriptions.

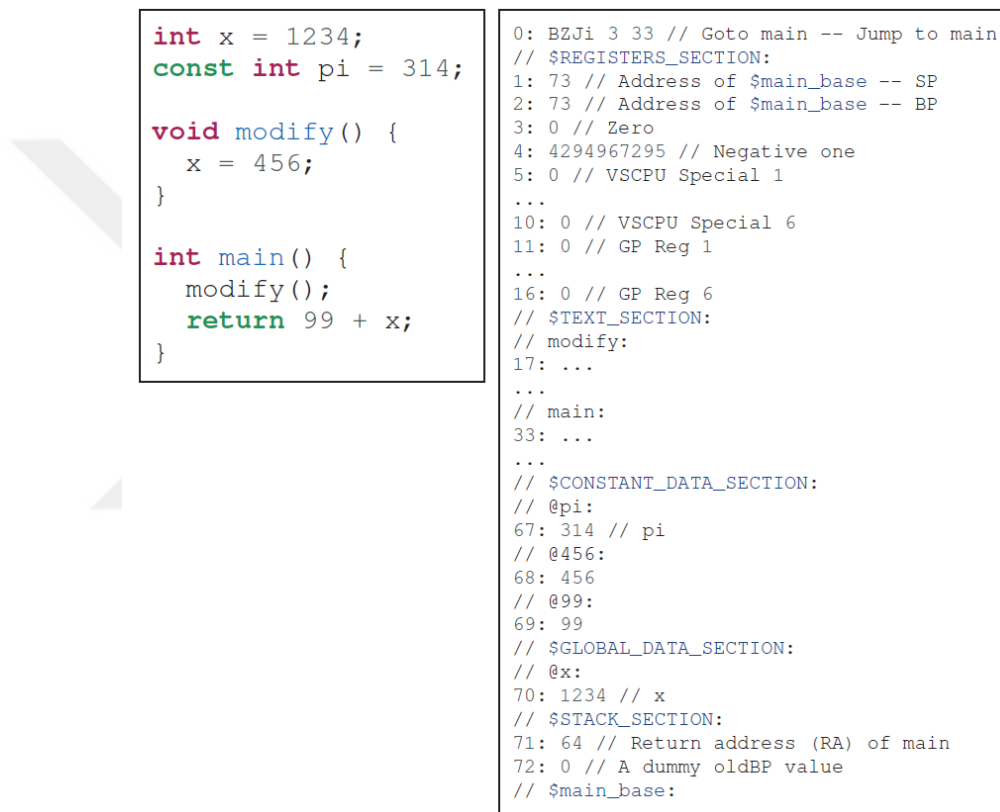


Figure 10: Example C program and assembly code generated by C compiler [10]

Table 1: Instruction word structure of VSCPU [10]

	Instruction Word			
bit position	[31:29]	[28]	[27:14]	[13:0]
field name	Opcode	immediate	operand A	operand B
bit width	3-bit	1-bit	14-bit	14-bit

Table 2: Instruction set of VSCPU [10]

Mnemonic	Description	Microoperation
ADD A B	unsigned add	$*A = *A + *B$
ADDi A B	unsigned add immediate	$*A = *A + B$
NAND A B	bitwise NAND	$*A = \sim (*A \& *B)$
NANDi A B	bitwise NAND immediate	$*A = \sim (*A \& B)$
SRL A B	shift right/left	$*A = (*B < 32) ? (*A >> *B) : (*A << (*B - 32))$
SRLi A B	shift right/left immediate	$*A = (B < 32) ? (*A >> B) : (*A << (B - 32))$
LT A B	compare and set	$*A = (*A < *B) ? 1 : 0$
LTi A B	compare and set immediate	$*A = (*A < B) ? 1 : 0$
MUL A B	unsigned multiply	$*A = *A * *B$
MULi A B	unsigned multiply immediate	$*A = *A * B$
CP A B	copy data	$*A = *B$
CPi A B	copy data immediate	$*A = B$
CPI A B	copy data indirect	$*A = *(*B)$
CPIi A B	copy data indirect immediate	$ *(*A) = *B$
BZJ A B	branch on zero	$PC = (*B == 0) ? *A : (PC + 1)$
BZJi A B	unconditional branch	$PC = *A + B$

In the VSCPU toolchain, there is a Clang-based C compiler supporting a small subset of C syntax. C programs written for VSCPU are processed and analyzed; errors in the programs are detected and shown. The C compiler generates an assembly code from an appropriate C program written for the VSCPU compiler. The instruction set simulator is Python-based and it can simulate hand-written or compiler generated

assembly programs for VSCPU. The assembler is also Python-based and it can convert VSCPU assembly code to VSCPU machine code which may be used by inserting it to VSCPU data and instruction memories. In Figure 10, a sample C program and its assembly code generated by the C compiler can be seen.

4.4.1 Design Details

The C-based design of the PRESENT algorithm from the FELICS project is used here as in the HLS implementation. The C code is compiled with the C compiler of the VSCPU and the Assembly code is extracted. Then, using VSCPU assembler, instruction and data memories are generated. Memory files as RTL designs are integrated RTL design of VSCPU and the complete design is simulated on Xilinx software. Finally, the functionality of the algorithm with VSCPU implementation is verified and the design is synthesized and implemented.

4.4.2 Design Results

As a result, 2210 32-bit instruction memory, which is stored in a ROM, and 58 32-bit data memory, which is stored in a RAM, were needed according to assembler generated memory files. The result of encryption operation is computed at the end of 557997 clock cycles and the design can work with a clock frequency of up to 67 MHz using 8966 LUTs and 3203 FFs.

4.5 With Customized VSCPU

The PRESENT block cipher algorithm written in C program can be compiled to the assembly code by the C compiler of VSCPU. However, the output code is not optimized enough to have fewer instructions. Operations in the PRESENT algorithm are not

computationally complex but the compiled Assembly code is too long. Therefore, modifying the instruction set and the processor architecture, and writing the assembly code manually is become a necessity in order to decrease the area and resource consumption and the latency for this specific algorithm. This would lead also to energy efficient design of the PRESENT block cipher. Instruction set simulator and the assembler is also changed for the compliance with the new customized VSCPU.

4.5.1 Design Details (VSCPU-80)

Due to the lack of XOR operation and algorithm specific commands such as SBOX in the VSCPU's instruction set, the compiled Assembly code consumed a lot of memory and took too long to compute an encryption operation. Therefore, to improve performance and reduce resource consumption, we changed the instruction set of the VSCPU and wrote the Assembly code manually.

The block length of the PRESENT algorithm is 64 bits, and the key length is 80 bits. For every round operation 3 functions are executed on 64-bit blocks and another 3 functions are executed on 80-bit key value for key schedule operations. To minimize the latency and the program memory for efficiency, the data path length of the VSCPU is increased to 80 bits. With that modification, the size of the program memory is highly reduced as you can see in Table 6; only 3 instructions in program memory are enough for round operations and another 3 instructions for key schedule operation.

We also added five new instructions in the algorithm that implement substitution (SBOX, SBOX64), left rotation (RRL), permutation (PER), and XOR functions. We removed instructions that were not used in the algorithm from the instruction set and the

new instruction set is described in Table 3. The handwritten Assembly code is also shown in Table 4.

Table 3: Instruction set of VSCPU-80

Mnemonic	Description	Microoperation
ADD A B	unsigned add	$*A = *A + *B$
SRLi A B	shift right by 16	$*A = *A \gg 16$
CP A B	copy data	$*A = *B$
BZJ A B	branch on zero	$PC = (*B == 0) ? *A : (PC + 1)$
BZJi A B	unconditional branch	$PC = *A + B$
XOR A B	bitwise XOR	$*A = *A \wedge *B$
RRLi A B	rotate right by 15 or 4	$*A = (B == 4) ? (*A \text{ rot } 4) : (*A \text{ rot } 15)$
SBOX A B	Substitution of LSB 4-bit	$*A = \{ *A[79:4], \text{SBOX}(*A[3:0]) \}$
SBOX64 A B	substitution of LSB 64-bit	$*A = \{ *A[79:64], \text{SBOX}(*A[63:60]), \text{SBOX}(*A[59:56]), \dots, \text{SBOX}(*A[3:0]) \}$
PER A B	permutation of LSB 64-bit	$*A = \{ *A[79:64], \text{PER}(*A[63:0]) \}$

Another change in the VSCPU was the transition from a combined memory unit in the Von Neumann architecture to the separate storage of data and instructions, as in the Harvard architecture. The aim was to take advantage of the fact that the bit length of the instructions is shorter than the bit length of the data and to consume fewer resources.

Table 4: Handwritten assembly code of PRESENT algorithm for VSCPU-80

Program Memory	Data Memory
<pre>// state = state xor rk 0: CP 202 201 1: SRLi 202 16 2: XOR 200 202 // jump halt if counter == 32*32768 3: CP 202 203 4: ADD 202 205 5: BZJ 204 202 // key schedule 6: RRLi 201 15 7: SBOX 201 201 8: RRLi 201 4 9: ADD 203 206 10: XOR 201 203 // encrypt - SBox 11: SBOX64 200 200 // encrypt - PLayer 12: PER 200 200 // jump to the beginning for 32 rounds 13: BZJi 207 0 // Goto \$HALT - Terminate process 14: BZJi 207 20</pre>	<pre>// Data registers 200: 0 // state 201: 0 // key 202: 0 // temp 203: 0 // counter 204: 14 // halt address 205: 1208925819614629173690368 // = -32*32768 206: 32768 // increase counter with that 207: 0 // Zero</pre>

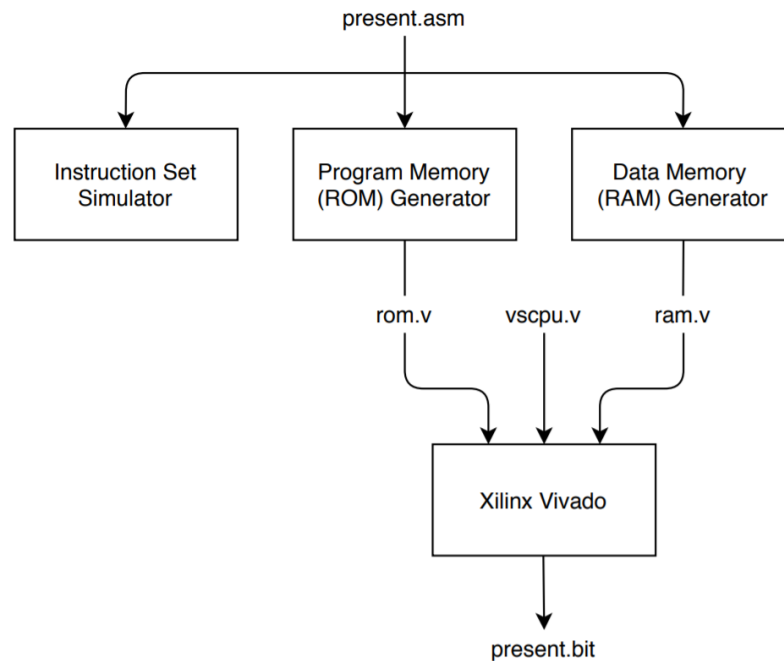


Figure 11: Design flow of the customized VSCPU

Figure 11 shows the design flow of the customized VSCPU. At first, an assembly program is written for the modified instruction set. This assembly program is simulated using the instruction set simulator, which is obtained by modifying the original instruction set simulator according to the new instruction set. If the assembly program is working correctly on the simulator, program memory and data memory should be generated with ROM and RAM generator scripts. The resulting files and the RTL design of the customized VSCPU is synthesized and implemented using Xilinx Vivado software.

4.5.2 Design Results (VSCPU-80)

This design required 15 instruction words of 10-bit length and 8 data words of 80-bit length. The design, which completes an encryption operation in 502 cycles, can work with up to 141 MHz clock frequency, using 1023 LUTs and 507 FFs.

4.5.3 Optimized Design Details (VSCPU-16)

To further reduce the resource consumption of the VSCPU with the 80-bit data path (VSCPU-80) design, we designed a VSCPU with the 16-bit data path (VSCPU-16). As shown in Table 5, we have made significant changes to the instruction set.

In VSCPU-16 design, an additional register named “W” has been defined to execute the algorithm in a more optimized manner. Since our key value is 80-bit, it is stored in 5 separate data addresses in this 16-bit architecture design. A new instruction called SRW3 was created, which also uses the W register to perform the 3-bit right shift required in the key schedule function. Using this instruction and W register repeatedly, rotation of the key value, which is found in different addresses, can be made easily.

With the new SBOX16 instruction, the parameter 0 provides the SBOX conversion in the key schedule function only to a 4-bit section, and with the parameter 1, all input values are converted through SBOX operation.

The permutation function is also provided by calling the PER0, PER1, PER2, and PER3 instructions twice in succession. The work in [23] is used to create these permutation operation instructions.

Table 5: Instruction set of VSCPU-16

Mnemonic	Description	Microoperation
ADD A B	unsigned add	$*A = *A + *B$
CP A B	copy data	$*A = *B$
BZJ A B	branch on zero	$PC = (*B == 0) ? *A : (PC + 1)$
BZJi A B	unconditional branch	$PC = *A + B$
XOR A B	bitwise XOR	$*A = *A \wedge *B$
GETW A B	set A to W	$*A = (B == 0) ? *A : W$ $W = 0$
SRW3 A B	rotate by 3	$temp = *A \ll 13$ $*A = (*A \gg 3) \mid W$ $W = temp$
SBOX16 A B	substitution	$*A = (B == 0) ? \{SBOX(*A[15:12]), *A[11:0]\} : \{SBOX(*A[15:12]), SBOX(*A[11:8]), SBOX(*A[7:4]), SBOX(*A[3:0])\}$
PER3 A B	permutation	$W = \{ *A[15], *A[11], *A[7], *A[3], *B[15], *B[11], *B[7], *B[3], W[15:8] \}$
PER2 A B	permutation	$W = \{ *A[14], *A[10], *A[6], *A[2], *B[14], *B[10], *B[6], *B[2], W[15:8] \}$
PER1 A B	permutation	$W = \{ *A[13], *A[9], *A[5], *A[1], *B[13], *B[9], *B[5], *B[1], W[15:8] \}$
PER0 A B	permutation	$W = \{ *A[12], *A[8], *A[4], *A[0], *B[12], *B[8], *B[4], *B[0], W[15:8] \}$

4.5.4 Optimized Design Results (VSCPU-16)

With the transition from 80-bit architecture to 16-bit architecture and a new instruction set, 45 instruction words of 12-bit and 16 data words of 16-bit is needed, and the algorithm computes the result in 1402 cycles. However, the new design can operate with a clock frequency of up to 153 MHz and consumes 270 LUTs and 291 FFs.

4.6 Comparison of Our Implementations and Other Works

In this work, the area and power/energy efficient design of the PRESENT block cipher algorithm is aimed. Pure RTL, Vivado HLS, PicoBlaze, VSCPU, and customized VSCPU are used as implementation methods. This algorithm is also implemented on FPGAs by other teams. The most recent and promising researches are examined, and the results are compared with our implementation results. The implementation results are shown in Table 6 and Table 7.

In our work, area and energy/power efficient design is aimed. Also, energy and power consumption of an IC is highly related with silicon area (or hardware resource) consumption. Therefore, total count of Look-Up Tables (LUTs) and Flip-Flops (FFs) in FPGA implementations is the most important feature for comparison. Latency is also used for comparison.

With the pure RTL design, the best results were obtained in terms of performance and area consumption, and these results were used for comparison with other implementations. The design to be done in this way has disadvantages such that the design is coded and validated manually so that the design time may be a bit longer

than other implementations and that the design does not offer any flexibility to the lack of programmability.

Table 6: Memory related information on processor based implementations

Memory Information	PicoBlaze	VSCPU	VSCPU-80	VSCPU-16	[26]
Instruction Bit Length	18	32	10	12	8
Number of Instructions in the Program	509	2210	15	45	156
Data Bit Length	8	32	80	16	8
Number of Data in the Program	N/A	58	8	16	48

Table 7: Comparison of PRESENT implementations on FPGA

Consumption and Performance Information	Pure RTL	Vivado HLS	Pico Blaze	VS CPU	VS CPU-80	VS CPU-16	[24]	[25]	[26]
Look Up Tables (LUT)	213	466	292	8966	1023	270	222	226	377
Flip Flops (FFs)	213	765	2280	3203	507	291	201	89	N/A
Max. Clock Frequency in MHz	368	125	185	67	141	153	236	172	542
Latency in Cycles	32	29125	359943	557997	2008	5608	295	132	20030
Latency in Time	86,9 ns	233 us	1,9 ms	8,3 ms	14,2 us	36,6 us	1,2 us	767 ns	36,9 us
Latency x (LUTs + FFs)	37 us	286 ms	4,88 s	101,2 s	21,7 ms	20,5 ms	528 us	241 us	27,8 ms

Designing with HLS significantly reduces design time. However, when the obtained results are examined, it is seen that resource consumption increases approximately twice as compared to the pure RTL implementation. Besides, an

encryption operation results in 32 cycles in the pure RTL implementation, while 29125 cycles in the Vivado HLS implementation. Considering the maximum clock frequencies that the designs can operate, it is seen that there is an increase of approximately 2681 times in latency. Moreover, as with the pure RTL design, the design produced by Vivado HLS does not include a processor, so it does not allow software development.

By using PicoBlaze, 5 times more resources are used than pure RTL design. When the clock frequencies and the number of cycles required for an encryption operation are considered, PicoBlaze implementation is approximately 22324 times slower than the pure RTL design. Therefore, this implementation would consume very high energy for an encryption operation. However, due to its processor based architecture, software development on this implementation is possible.

Considering the resulting resource consumption and performance, we can say that VSCPU implementation has the worst results among other options. The reasons for this may be that the instruction set and architecture of VSCPU are not well suited to the PRESENT algorithm and that the C code compiler does not produce good results. However, this implementation has the advantages that it provides a much faster design opportunity than a pure RTL implementation and allows for software development due to its processor based structure.

When the customized VSCPU design with 80-bit datapath is evaluated in terms of performance, it provides much higher performance than other realizations except pure RTL implementation. So, modifying instruction set and processor architecture for non-complex operations proved to be useful for better performance. However, this

implementation consumes high resource consumption; it only gives better results than VSCPU and PicoBlaze in terms of resource consumption.

The resource consumption of customized VSCPU with 16-bit datapath is less than Vivado HLS, PicoBlaze, VSCPU, and VSCPU-80 implementations. Besides, it consumes only 31% more resources than pure RTL implementation. In terms of latency, pure RTL implementation is far better than VSCPU-16.

In the end, VSCPU-16 and Pure RTL implementations provide better results in terms of resource consumption, latency, and power/energy consumption than Vivado HLS, PicoBlaze, VSCPU, and VSCPU-80 implementations. VSCPU-16 and Pure RTL implementations will be examined in the next section on a use case scenario.

In 2015, Tay et al. developed an 8-bit hardware architecture of the PRESENT algorithm on a Virtex-V FPGA [24]. They managed to reduce the number of resources for SBoxes due to 8-bit datapath, and Karnaugh mapping and further factorization of SBoxes. The resultant implementation consumes 222 LUTs and 201 FFs and it also can work with a clock frequency up to 236 MHz and it can compute a result in 295 cycles.

In 2016, Lara-Nino et al. created an architecture based on 16-bit datapath [25]. By doing that, the implementation of PRESENT on Spartan-6 FPGAs consumes only 226 LUTs and 89 FFs and the design works with a clock frequency up to 172 MHz and with a latency of 132 cycles.

Diehl et al. presented the implementation of 6 different ciphers including PRESENT using both custom hardware design and software design with an 8-bit microprocessor [26]. There are only 30 native instructions on this soft microprocessor and the data words and instruction words are 8-bit length. The processor is tailored to

the algorithms and unrequired functionality is removed before implementation as in our work. PRESENT implementation on Kintex-7 FPGAs using this custom processor consumes 377 LUTs, achieves a clock frequency up to 542 MHz, and takes 20030 cycles to complete an encryption operation. 156 instruction words and 48 data words are used in the implementation.

In 2013, Engels et al. customized a processor through the instruction set extension for efficient block cipher implementations [27]. PRESENT cipher is one of the implemented ciphers in that work. It has a latency of 6017 clock cycles and 3248 bit flash memory is utilized in addition to the logic area.

In conclusion, our pure RTL implementation consumes a similar amount of FPGA resources with [24] and [25]. However, the latency of [24] is almost 13 times more than the latency of our pure RTL implementation and the latency of [25] is almost 8 times more than the latency of our pure RTL implementation. Moreover, our VSCPU design with 16-bit architecture is more efficient and consumes lower resources than [26] and [27]. Our results were also published in our paper [5].

4.7 A Use Case Scenario for PRESENT in an IoT Application

In a real world application of the secure IoT devices, there are many more operations which should be done other than encryption or decryption. For example, data gathered from sensors may be pre-processed, some analysis may be needed for taking decisions, sensors may need to be calibrated or adjusted, communication protocols may need to be implemented. Therefore, the whole functionality of an IoT device should be taken into account when designing an area and energy efficient IC for IoT.

Using processors is the best way to provide the functionality of IoT devices since they can enable short design time, they provide flexibility for later modification in the functionality and they consume fewer resources when the same micro-operations are used for different parts of the algorithms that are implemented on IoT devices. Due to these advantages, processor based designs are compared.

In the previous section, we compared PRESENT block cipher implementations using different platforms. It is seen that two of these implementations provide better results in terms of area, latency, and power/energy consumption: Pure RTL and VSCPU-16 implementation of the PRESENT block cipher algorithm. In this section, a simple use case scenario example will be used for comparing these implementations. In this use case scenario, 4 sensor data is taken and applied a simple pre-processing before encryption.

4.7.1 Design Details

As the pre-processing, the following simple algorithm is used. At first, sensor values are sampled from 4 different sensors and they are saved into 4 different memory registers. Then, sensor values are resampled again, and they are saved into other memory locations. The last sampled values are divided by 8 and this operation is completed simply by shifting the last four sensor values to the left by 3. Finally, previous sensor values and last sensor values, that are divided by 8, are added together and the logical value of the LSB part of the result is reversed by applying XOR operation with 0xffff. So, the pre-processing is completed. After that, the result is encrypted with the PRESENT block cipher algorithm.

There are 4 different instructions are used in the pre-processing algorithm: CP, SRL, ADD, and XOR. This algorithm is chosen to have the same instructions with VSCPU-16 implementation of the PRESENT algorithm since we claim that the processors consume fewer resources when the different parts of the algorithm use the same instructions. As said before, processor based designs are used for comparison but the PRESENT block cipher implementation is different in these designs.

In both designs, the VSCPU-16 processor is used as a base. In Design 1, the PRESENT cipher is implemented as Pure RTL design and its inputs and outputs are mapped to some parts of the data memory of the VSCPU-16. The instruction set is reduced by removing the following unnecessary instructions: GETW, SBOX, PER0, PER1, PER2, PER3. Also, the assembly program is rewritten for Design 1. In Design 2, the PRESENT cipher is implemented as the assembly code as in section 4.5.3. So, there is no change in the instruction set and the processor architecture of VSCPU-16 for Design 2. Only, the assembly program is updated to include the pre-processing part of the algorithm.

4.7.2 Design Results

The implementation results are shown in Table 8. Design 2 has a longer Assembly program than Design 1 and Design 1 includes a cryptographic co-processor connected to the main processor as memory-mapped. Both designs can work with a clock frequency very close to each other. The second design consumes %40 fewer resources than the first design, but its latency is more than the first design. These results show that customizing a processor architecture and its instruction set may lead an area and energy efficient design.

Table 8: Implementation results of Design 1 and Design 2

Consumption and Performance Information	Design 1	Design 2
Number of instructions in the program	21	64
Number of data in the program	28	24
Look Up Tables (LUT)	409	285
Flip Flops (FF)	566	302
Max. Clock Frequency in MHz	172,4	171,8
Latency in Cycles	107	5799
Latency in Time	6,2 us	337,5 us
Latency x (LUTs + FFs)	6,04 ms	198,1 ms

CHAPTER V

CONCLUSIONS AND FUTURE WORK

The smart devices surrounding us produce solutions for daily life problems and increase our standard of living. The information security of these devices has become an important issue. Since there are some limitations for silicon area and power/energy consumption in the integrated circuits which are used in IoT devices, Lightweight Cryptography algorithms replaced modern cryptographic algorithms by decreasing processing complexity and the area usage of the chips while providing a reasonable amount of security.

A classic functionality of modern IoT devices may include pre-processing of the data gathered by sensors, encryption of the data, and sharing data with other devices. Ensuring this functionality with minimum energy, power, and area consumption on the integrated circuits is a challenging task and the main goal of this work. Also flexibility ie. the programmability of the design is important since it allows shorter design time and later modification of the functionality. For this purpose, at first, a popular lightweight block cipher algorithm PRESENT is implemented with pure RTL coding, Vivado HLS, PicoBlaze, and VSCPU. Then, the instruction set and the architecture of the VSCPU is customized for an optimized design. Finally, an example algorithm as a complete usecase in IoT applications is implemented in two different ways: VSCPU with PRESENT co-processor and instruction set extended VSCPU for PRESENT. The

results are compared for all implementations with each other and with most recent and promising researches.

As can be seen in the previous chapter, the customized VSCPU-16 implementation, which has been modified by changing the instruction set and the architecture, is the most optimized solution due to its low resource consumption, acceptable latency, low power/energy consumption, and the ability to develop software on it. Pure RTL implementation of PRESENT has the best results among our implementations in terms of latency, area, and power/energy consumption. However, a complete IoT product needs a processor anyway for pre-processing of gathered data and communication protocols. We show that the customized processor approach consumes a smaller area even compared to the pure RTL approach through the implementation of PRESENT in a complete usecase.

The other popular lightweight block cipher algorithms such as PRINCE, SIMON, SPECK *etc.* have similar structures to PRESENT block cipher. They consist of rounds and key schedule operations. Shift, Rotate, XOR, Add, and SBOX micro-operations are used in these round and key schedule operations. Therefore, our customization processor methodology may be applied to other lightweight cryptographic algorithms for area and power/energy efficient implementations.

This thesis also contributes to the literature with the implementation of PRESENT in various ways, namely, pure RTL, HLS, standard processor, customized processor approach, and by comparing these with the results in the literature.

Our VSCPU designs benefited from the MC2SC project by using some of its tools such as instruction set simulator, compiler, and assemblers. This thesis also

allowed us to contribute to the MC2SC project as a usecase, and the tools that resulted from the MC2SC project are improved in return.

In future studies, pipeline mechanism may be added to customized VSCPU. This will considerably improve the performance and decrease the latency of an encryption operation in return for a small amount of increase in resource consumption. VSCPU's compiler may also be developed according to the changing instruction set, and architecture, thus rapid prototyping can be realized completely. In this way, it would be possible to make energy and area efficient designs in a shorter time.

APPENDIX A

SOURCE CODE FOR DESIGN 1

In this section RTL design source codes of Design 1 (customized VSCPU-16 with PRESENT cipher as a co-processor) will be shared.

tb_vscpu_16_top.v:

```
`timescale 1ns / 1ps
module tb_vscpu_16_top;

reg clk, rst;
wire [15:0] s1_i, s2_i, s3_i, s4_i;
wire [63:0] ct_o;

assign s1_i = 11111;
assign s2_i = 22222;
assign s3_i = 33333;
assign s4_i = 44444;

initial begin
    clk = 0;
    forever
        #5 clk = ~clk;
end

initial begin
    rst = 1;
    repeat(5) @(negedge clk);

    rst <= #1 0;
    #120000;
end

vscpu_16_top uut_vscpu_16_top (.clk(clk), .rst(rst), .s1_i(s1_i), .s2_i(s2_i),
.s3_i(s3_i), .s4_i(s4_i), .ct_o(ct_o));

endmodule
```

vscpu_16_top.v:

```
`timescale 1ns / 1ps

module vscpu_16_top(clk, rst, s1_i, s2_i, s3_i, s4_i, ct_o);

parameter INSTR_ADDR_SIZE = 6, INSTR_SIZE = 14;
parameter DATA_ADDR_SIZE = 5, DATA_SIZE = 16;
parameter OPSIZE = 4;

input clk, rst;
input [15:0] s1_i, s2_i, s3_i, s4_i;
output [63:0] ct_o;

wire [INSTR_ADDR_SIZE-1:0] instr_addr_o;
wire [INSTR_SIZE-1:0] instr_i;
wire [DATA_ADDR_SIZE-1:0] data_addr_o;
wire [DATA_SIZE-1:0] data_i;
wire [DATA_SIZE-1:0] data_o;
wire data_wr_en;

wire [63:0] p_pt, p_ct;
wire [79:0] p_key;
wire p_start, p_done;

instr_mem #(.INSTR_ADDR_SIZE(INSTR_ADDR_SIZE),
.INSTR_SIZE(INSTR_SIZE)) uut_rom(.instr_addr_i(instr_addr_o),
.instr_o(instr_i));

data_mem #(.DATA_ADDR_SIZE(DATA_ADDR_SIZE),
.DATA_SIZE(DATA_SIZE)) uut_ram(.clk(clk), .rst(rst),
.data_addr_i(data_addr_o), .data_o(data_i), .data_i(data_o),
.data_wr_en(data_wr_en), .s1_i(s1_i), .s2_i(s2_i), .s3_i(s3_i), .s4_i(s4_i),
.ct_o(ct_o), .p_pt(p_pt), .p_key(p_key), .p_start(p_start), .p_ct(p_ct),
.p_done(p_done));

vscpu_16 #(.INSTR_ADDR_SIZE(INSTR_ADDR_SIZE),
.INSTR_SIZE(INSTR_SIZE), .DATA_ADDR_SIZE(DATA_ADDR_SIZE),
.DATA_SIZE(DATA_SIZE), .OPSIZE(OPSIZE)) uut_vscpu_16 (.clk(clk),
.rst(rst), .instr_addr_o(instr_addr_o), .instr_i(instr_i), .data_addr_o(data_addr_o),
.data_i(data_i), .data_o(data_o), .data_wr_en(data_wr_en));

present_top present(.clk(clk), .plaintext(p_pt), .key(p_key), .start(p_start),
.ciphertext(p_ct), .done(p_done));

endmodule
```

vscpu_16.v:

```
`timescale 1ns / 1ps

`define CP 4'b0000
`define SRLi 4'b0001
`define ADD 4'b0010
`define XOR 4'b0011
`define BZJ 4'b0100

module vscpu_16(clk, rst, instr_addr_o, instr_i, data_addr_o, data_i, data_o,
data_wr_en);

parameter INSTR_ADDR_SIZE = 6, INSTR_SIZE = 14;
parameter DATA_ADDR_SIZE = 5, DATA_SIZE = 16;
parameter OPSIZE = 4;

input clk,rst;
output reg [INSTR_ADDR_SIZE-1:0] instr_addr_o;
input [INSTR_SIZE-1:0] instr_i;
output reg [DATA_ADDR_SIZE-1:0] data_addr_o;
input [DATA_SIZE-1:0] data_i;
output reg [DATA_SIZE-1:0] data_o;
output reg data_wr_en;

reg [1:0] state,stateNext;
reg [INSTR_ADDR_SIZE-1:0] instr_addr_reg;
reg [INSTR_ADDR_SIZE-1:0] pCounter, pCounterNext;
reg [DATA_SIZE-1:0] r1, r1Next, r2, r2Next;
reg [OPSIZE-1:0] opCode, opCodeNext;
reg [DATA_ADDR_SIZE-1:0] a_addr, a_addr_next, b_addr, b_addr_next;

reg [DATA_SIZE-1:0] Wreg, WNext;

always@(posedge clk) begin
    instr_addr_reg <= instr_addr_o;
    pCounter <= pCounterNext;
    state <= stateNext;
    opCode <= opCodeNext;
    a_addr <= a_addr_next;
    b_addr <= b_addr_next;
    r1 <= r1Next;
    r2 <= r2Next;
    Wreg <= WNext;
end

always@(*) begin
    WNext = Wreg;
    instr_addr_o = instr_addr_reg;
    data_addr_o = 0;
end
```

```

data_o = 0;
data_wr_en = 0;
pCounterNext = pCounter;
opCodeNext = opCode;
a_addr_next = a_addr;
b_addr_next = b_addr;
stateNext = state;
r1Next = r1;
r2Next = r2;
if(rst)begin
    pCounterNext = 0;
    stateNext = 0;
    opCodeNext = 0;
    a_addr_next = 0;
    b_addr_next = 0;
    data_wr_en = 0;
    r1Next = 0;
    r2Next = 0;
    WNext = 0;
end else begin
    case(state)
        0:begin
            instr_addr_o = pCounter;
            stateNext = 1;
        end
        1:begin
            opCodeNext = instr_i[INSTR_SIZE-1:INSTR_SIZE-
OPSIZE];
            a_addr_next =
instr_i[DATA_ADDR_SIZE+DATA_ADDR_SIZE-1:DATA_ADDR_SIZE];
            b_addr_next = instr_i[DATA_ADDR_SIZE-1:0];
            data_addr_o = a_addr_next;
            stateNext = 2;
        end
        2:begin
            r1Next = data_i;
            data_addr_o = b_addr;
            stateNext = 3;
        end
        3: begin
            r2Next = data_i;
            case (opCode)
                `ADD:begin
                    data_wr_en = 1;
                    data_addr_o = a_addr;
                    data_o = r1 + r2Next;
                    pCounterNext = pCounter + 1;
                    stateNext = 0;
                end
                `SRLi:begin

```

```

data_wr_en = 1;
data_addr_o = a_addr;
data_o = (r1 >> 3) | Wreg;
WNext = (r1 << (DATA_SIZE-3));
pCounterNext = pCounter + 1;
stateNext = 0;
end
`CP:begin
data_wr_en = 1;
data_addr_o = a_addr;
data_o = r2Next;
pCounterNext = pCounter + 1;
stateNext = 0;
end
`BZJ:begin
data_wr_en = 0;
pCounterNext = (r2Next == 0) ? r1 :
(pCounter + 1);
stateNext = 0;
end
`XOR:begin
data_wr_en = 1;
data_addr_o = a_addr;
data_o = (r1 ^ r2Next);
pCounterNext = pCounter + 1;
stateNext = 0;
end
endcase
end
endcase
end
end
endmodule

```

instr_mem.v:

```

`timescale 1ns / 1ps

// CP = 4'b0000
// SRLi = 4'b0001
// ADD = 4'b0010
// XOR = 4'b0011
// BZJ = 4'b0100

module instr_mem(instr_addr_i, instr_o);

parameter INSTR_ADDR_SIZE = 6, INSTR_SIZE = 14;

```

```

input [INSTR_ADDR_SIZE-1:0] instr_addr_i;
output reg [INSTR_SIZE-1:0] instr_o;

always @(*) begin
    case (instr_addr_i)
        0: instr_o = 14'b00001011100000;
            1: instr_o = 14'b00001100000001;
            2: instr_o = 14'b00001100100010;
            3: instr_o = 14'b00001101000011;
            4: instr_o = 14'b00000001000000;
            5: instr_o = 14'b000000010100001;
            6: instr_o = 14'b000000011000010;
            7: instr_o = 14'b000000011100011;
            8: instr_o = 14'b00010010000011;
            9: instr_o = 14'b00010010100011;
            10: instr_o = 14'b00010011000011;
            11: instr_o = 14'b00010011100011;
            12: instr_o = 14'b00100010010111;
            13: instr_o = 14'b00100010111000;
            14: instr_o = 14'b00100011011001;
            15: instr_o = 14'b00100011111010;
            16: instr_o = 14'b00110010011011;
            17: instr_o = 14'b00001000110101;
            18: instr_o = 14'b00001000110100;
            19: instr_o = 14'b01001011010010;
            20: instr_o = 14'b01001001110100;
            default: instr_o = 0;
    endcase
end

endmodule

```

data_mem.v:

```

`timescale 1ns / 1ps
module data_mem(clk, rst, data_addr_i, data_o, data_i, data_wr_en,
                s1_i, s2_i, s3_i, s4_i, ct_o, p_pt, p_ct, p_key, p_start, p_done);

    parameter DATA_ADDR_SIZE = 5;
    parameter DATA_SIZE = 16;
    parameter RAM_DEPTH = 28;

    input clk, rst, data_wr_en;
    input [DATA_ADDR_SIZE-1:0] data_addr_i;
    input [DATA_SIZE-1:0] data_i;
    output reg [DATA_SIZE-1:0] data_o;
    input [15:0] s1_i, s2_i, s3_i, s4_i;
    output [63:0] ct_o;

```

```

output wire [63:0] p_pt;
input wire [63:0] p_ct;
output wire [79:0] p_key;
output wire p_start;
input wire p_done;

reg [DATA_SIZE-1:0] memory[0:RAM_DEPTH-1];
reg [DATA_SIZE-1:0] memoryNext[0:RAM_DEPTH-1];

assign ct_o = {memory[16], memory[15], memory[14], memory[13]};
assign p_pt = {memory[7], memory[6], memory[5], memory[4]};
assign p_key = {memory[12], memory[11], memory[10], memory[9],
memory[8]};
assign p_start = memory[17][0];

always @(posedge clk) begin
    data_o <= #1 memory[data_addr_i];

    memory[4] <= #1 memoryNext[4];
    memory[5] <= #1 memoryNext[5];
    memory[6] <= #1 memoryNext[6];
    memory[7] <= #1 memoryNext[7];
    memory[8] <= #1 memoryNext[8];
    memory[9] <= #1 memoryNext[9];
    memory[10] <= #1 memoryNext[10];
    memory[11] <= #1 memoryNext[11];
    memory[12] <= #1 memoryNext[12];
    memory[17] <= #1 memoryNext[17];
    memory[23] <= #1 memoryNext[23];
    memory[24] <= #1 memoryNext[24];
    memory[25] <= #1 memoryNext[25];
    memory[26] <= #1 memoryNext[26];

end

always @(*) begin
    memory[0] = s1_i;
    memory[1] = s2_i;
    memory[2] = s3_i;
    memory[3] = s4_i;
    memory[13] = p_ct[15:0];
    memory[14] = p_ct[31:16];
    memory[15] = p_ct[47:32];
    memory[16] = p_ct[63:48];
    memory[18] = {15'b0, p_done};
    memory[19] = 20;
    memory[20] = 0;
    memory[21] = 1;
    memory[22] = 19;
    memory[27] = 65535;

```

```

end

always @(*) begin

    if(rst) begin
        memoryNext[4] = 0;
        memoryNext[5] = 0;
        memoryNext[6] = 0;
        memoryNext[7] = 0;
        memoryNext[8] = 0;
        memoryNext[9] = 1;
        memoryNext[10] = 2;
        memoryNext[11] = 3;
        memoryNext[12] = 4;
        memoryNext[17] = 0;
        memoryNext[23] = 0;
        memoryNext[24] = 0;
        memoryNext[25] = 0;
        memoryNext[26] = 0;
    end else begin
        memoryNext[4] = memory[4];
        memoryNext[5] = memory[5];
        memoryNext[6] = memory[6];
        memoryNext[7] = memory[7];
        memoryNext[8] = memory[8];
        memoryNext[9] = memory[9];
        memoryNext[10] = memory[10];
        memoryNext[11] = memory[11];
        memoryNext[12] = memory[12];
        memoryNext[17] = memory[17];
        memoryNext[23] = memory[23];
        memoryNext[24] = memory[24];
        memoryNext[25] = memory[25];
        memoryNext[26] = memory[26];

        if(data_wr_en) begin
            memoryNext[data_addr_i] = data_i;
        end
    end

end

endmodule

```

present.asm:

```
// pre-processing
// copy previous sensor values
0: CP 223 200
1: CP 224 201
2: CP 225 202
3: CP 226 203

// copy next sensor values
4: CP 204 200
5: CP 205 201
6: CP 206 202
7: CP 207 203

// divide next sensor values by 8
8: SRLi 204 3
9: SRLi 205 3
10: SRLi 206 3
11: SRLi 207 3

// sum sensor values
12: ADD 204 223
13: ADD 205 224
14: ADD 206 225
15: ADD 207 226

// reverse lsb 16 bit of the sum
16: XOR 204 227

// start
17: CP 217 221
18: CP 217 220

// wait for done
19: BZJ 222 218

// Goto $HALT -- Instruction jumps to itself to terminate the process.
20: BZJ 219 220

// Data registers
// sensor inputs
200: 11111 // s1_i
201: 22222 // s2_i
202: 33333 // s3_i
203: 44444 // s4_i
// plaintext = msb[3 2 1 0]lsb
204: 0 // pt_0
205: 0 // pt_1
206: 0 // pt_2
```

```
207: 0 // pt_3
// key = msb[4 3 2 1 0]lsb
208: 0 // key_0
209: 1 // key_1
210: 2 // key_2
211: 3 // key_3
212: 4 // key_4
// ciphertext = msb[3 2 1 0]lsb
213: 0 // ct_0
214: 0 // ct_1
215: 0 // ct_2
216: 0 // ct_3
217: 0 // start
218: 0 // done
219: 20 // halt address
220: 0 // Zero
221: 1 // One
222: 19 // loop address
// temporary registers
223: 0 // temp_0
224: 0 // temp_1
225: 0 // temp_2
226: 0 // temp_3
227: 65535 // ffff
```

APPENDIX B

SOURCE CODE FOR DESIGN 2

In this section RTL design source codes of Design 2 (customized VSCPU-16) will be shared.

tb_vscpu_16_top.v:

```
`timescale 1ns / 1ps
module tb_vscpu_16_top;

reg clk, rst;
wire [15:0] s1_i, s2_i, s3_i, s4_i;
wire [63:0] ct_o;

assign s1_i = 11111;
assign s2_i = 22222;
assign s3_i = 33333;
assign s4_i = 44444;

initial begin
    clk = 0;
    forever
        #5 clk = ~clk;
end

initial begin
    rst = 1;
    repeat(5) @(negedge clk);

    rst <= #1 0;
    #120000;
end

vscpu_16_top uut_vscpu_16_top (.clk(clk), .rst(rst), .s1_i(s1_i), .s2_i(s2_i),
.s3_i(s3_i), .s4_i(s4_i), .ct_o(ct_o));

endmodule
```

vscpu_16_top.v:

```
`timescale 1ns / 1ps

module vscpu_16_top(clk, rst, s1_i, s2_i, s3_i, s4_i, ct_o);

parameter INSTR_ADDR_SIZE = 6, INSTR_SIZE = 14;
parameter DATA_ADDR_SIZE = 5, DATA_SIZE = 16;
parameter OPSIZE = 4;

input clk, rst;
input [15:0] s1_i, s2_i, s3_i, s4_i;
output [63:0] ct_o;

wire [INSTR_ADDR_SIZE-1:0] instr_addr_o;
wire [INSTR_SIZE-1:0] instr_i;
wire [DATA_ADDR_SIZE-1:0] data_addr_o;
wire [DATA_SIZE-1:0] data_i;
wire [DATA_SIZE-1:0] data_o;
wire data_wr_en;

instr_mem #(.INSTR_ADDR_SIZE(INSTR_ADDR_SIZE),
.INSTR_SIZE(INSTR_SIZE)) uut_rom(.instr_addr_i(instr_addr_o),
.instr_o(instr_i));

data_mem #(.DATA_ADDR_SIZE(DATA_ADDR_SIZE),
.DATA_SIZE(DATA_SIZE)) uut_ram(.clk(clk), .rst(rst),
.data_addr_i(data_addr_o), .data_o(data_i), .data_i(data_o),
.data_wr_en(data_wr_en), .s1_i(s1_i), .s2_i(s2_i), .s3_i(s3_i), .s4_i(s4_i),
.ct_o(ct_o));

vscpu_16 #(.INSTR_ADDR_SIZE(INSTR_ADDR_SIZE),
.INSTR_SIZE(INSTR_SIZE), .DATA_ADDR_SIZE(DATA_ADDR_SIZE),
.DATA_SIZE(DATA_SIZE), .OPSIZE(OPSIZE)) uut_vscpu_16 (.clk(clk),
.rst(rst), .instr_addr_o(instr_addr_o), .instr_i(instr_i), .data_addr_o(data_addr_o),
.data_i(data_i), .data_o(data_o), .data_wr_en(data_wr_en));

endmodule
```

vscpu_16.v:

```
`timescale 1ns / 1ps

`define CP 4'b0000
`define SRLi 4'b0001
```

```

`define ADD 4'b0010
`define XOR 4'b0011
`define BZJ 4'b0100
`define GETW 4'b0101
`define SBOX 4'b0110
`define PER3 4'b0111
`define PER2 4'b1000
`define PER1 4'b1001
`define PER0 4'b1010

module vscpu_16(clk, rst, instr_addr_o, instr_i, data_addr_o, data_i, data_o,
data_wr_en);

parameter INSTR_ADDR_SIZE = 6, INSTR_SIZE = 14;
parameter DATA_ADDR_SIZE = 5, DATA_SIZE = 16;
parameter OPSIZE = 4;

input clk,rst;
output reg [INSTR_ADDR_SIZE-1:0] instr_addr_o;
input [INSTR_SIZE-1:0] instr_i;
output reg [DATA_ADDR_SIZE-1:0] data_addr_o;
input [DATA_SIZE-1:0] data_i;
output reg [DATA_SIZE-1:0] data_o;
output reg data_wr_en;

reg [1:0] state,stateNext;
reg [INSTR_ADDR_SIZE-1:0] instr_addr_reg;
reg [INSTR_ADDR_SIZE-1:0] pCounter, pCounterNext;
reg [DATA_SIZE-1:0] r1, r1Next, r2, r2Next;
reg [OPSIZE-1:0] opCode, opCodeNext;
reg [DATA_ADDR_SIZE-1:0] a_addr, a_addr_next, b_addr, b_addr_next;

reg [3:0] sbox [0:15];
reg [DATA_SIZE-1:0] Wreg, WNext;
reg Wadd, WaddNext;

always@(posedge clk) begin
    instr_addr_reg <= instr_addr_o;
    pCounter <= pCounterNext;
    state <= stateNext;
    opCode <= opCodeNext;
    a_addr <= a_addr_next;
    b_addr <= b_addr_next;
    r1 <= r1Next;
    r2 <= r2Next;
    Wreg <= WNext;
    Wadd <= WaddNext;
end

always@(*) begin

```

```

sbox[0] = 4'd12;
sbox[1] = 4'd5;
sbox[2] = 4'd6;
sbox[3] = 4'd11;
sbox[4] = 4'd9;
sbox[5] = 4'd0;
sbox[6] = 4'd10;
sbox[7] = 4'd13;
sbox[8] = 4'd3;
sbox[9] = 4'd14;
sbox[10] = 4'd15;
sbox[11] = 4'd8;
sbox[12] = 4'd4;
sbox[13] = 4'd7;
sbox[14] = 4'd1;
sbox[15] = 4'd2;
WNext = Wreg;
WaddNext = Wadd;
instr_addr_o = instr_addr_reg;
data_addr_o = 0;
data_o = 0;
data_wr_en = 0;
pCounterNext = pCounter;
opCodeNext = opCode;
a_addr_next = a_addr;
b_addr_next = b_addr;
stateNext = state;
r1Next = r1;
r2Next = r2;
if(rst)begin
    pCounterNext = 0;
    stateNext = 0;
    opCodeNext = 0;
    a_addr_next = 0;
    b_addr_next = 0;
    data_wr_en = 0;
    r1Next = 0;
    r2Next = 0;
    WNext = 0;
    WaddNext = 0;
end else begin
    case(state)
        0:begin
            instr_addr_o = pCounter;
            stateNext = 1;
        end
        1:begin
            opCodeNext = instr_i[INSTR_SIZE-1:INSTR_SIZE-
OPSIZE];
            a_addr_next =

```

```

instr_i[DATA_ADDR_SIZE+DATA_ADDR_SIZE-1:DATA_ADDR_SIZE];
    b_addr_next = instr_i[DATA_ADDR_SIZE-1:0];
    data_addr_o = a_addr_next;
    stateNext = 2;
end
2:begin
    r1Next = data_i;

    if (opCode == `SRLi || opCode == `SBOX || opCode
== `GETW)
        data_addr_o = a_addr;
    else
        data_addr_o = b_addr;
        stateNext = 3;
    end
3: begin
    r2Next = data_i;
    case (opCode)
        `ADD:begin
            data_wr_en = 1;
            data_addr_o = a_addr;

            data_o = r1 + r2Next;

            pCounterNext = pCounter + 1;
            stateNext = 0;
        end
        `SRLi:begin
            data_wr_en = 1;
            data_addr_o = a_addr;

            data_o = (r1 >> 3) | Wreg;
            WNext = (r1 << (DATA_SIZE-3));

            pCounterNext = pCounter + 1;
            stateNext = 0;
        end
        `CP:begin
            data_wr_en = 1;
            data_addr_o = a_addr;
            data_o = r2Next;
            pCounterNext = pCounter + 1;
            stateNext = 0;
        end
        `BZJ:begin
            data_wr_en = 0;
            pCounterNext = (r2Next == 0) ? r1 :
(pCounter + 1);

            stateNext = 0;
        end
        `XOR:begin
            data_wr_en = 1;
            data_addr_o = a_addr;
            data_o = (r1 ^ r2Next);

```

```

                                pCounterNext = pCounter + 1;
                                stateNext = 0;
                                end
                                `SBOX:begin
                                data_wr_en = 1;
                                data_addr_o = a_addr;
                                if (b_addr == 0) begin
                                    data_o = {sbox[r1[DATA_SIZE-1:DATA_SIZE-4]] ,
r1[DATA_SIZE-5:0]};
                                end else begin
                                    data_o = {sbox[r1[15:12]], sbox[r1[11:8]], sbox[r1[7:4]],
sbox[r1[3:0]]};
                                end
                                pCounterNext = pCounter + 1;
                                stateNext = 0;
                                end
                                `PER0:begin
                                data_wr_en = 0;
                                data_addr_o = a_addr;
                                WNext = {r1[12], r1[8], r1[4], r1[0], r2Next[12], r2Next[8],
r2Next[4], r2Next[0], Wreg[15:8]};
                                pCounterNext = pCounter + 1;
                                stateNext = 0;
                                end
                                `PER1:begin
                                data_wr_en = 0;
                                data_addr_o = a_addr;
                                WNext = {r1[13], r1[9], r1[5], r1[1], r2Next[13], r2Next[9],
r2Next[5], r2Next[1], Wreg[15:8]};
                                pCounterNext = pCounter + 1;
                                stateNext = 0;
                                end
                                `PER2:begin
                                data_wr_en = 0;
                                data_addr_o = a_addr;
                                WNext = {r1[14], r1[10], r1[6], r1[2], r2Next[14], r2Next[10],
r2Next[6], r2Next[2], Wreg[15:8]};
                                pCounterNext = pCounter + 1;
                                stateNext = 0;
                                end
                                `PER3:begin
                                data_wr_en = 0;
                                data_addr_o = a_addr;
                                WNext = {r1[15], r1[11], r1[7], r1[3], r2Next[15], r2Next[11],
r2Next[7], r2Next[3], Wreg[15:8]};
                                pCounterNext = pCounter + 1;
                                stateNext = 0;
                                end
                                `GETW:begin
                                data_wr_en = 1;

```

```

data_addr_o = a_addr;
if (b_addr == 0)
    WNext = 0;
else begin
    data_o = Wreg;
    WNext = 0;
end

pCounterNext = pCounter + 1;
stateNext = 0;

end
endcase
end
endcase
end
end
endmodule

```

instr_mem.v:

```

`timescale 1ns / 1ps

// CP = 4'b0000
// SRLi = 4'b0001
// ADD = 4'b0010
// XOR = 4'b0011
// BZJ = 4'b0100
// GETW = 4'b0101
// SBOX = 4'b0110
// PER3 = 4'b0111
// PER2 = 4'b1000
// PER1 = 4'b1001
// PER0 = 4'b1010

module instr_mem(instr_addr_i, instr_o);

parameter INSTR_ADDR_SIZE = 6, INSTR_SIZE = 14;

input [INSTR_ADDR_SIZE-1:0] instr_addr_i;
output reg [INSTR_SIZE-1:0] instr_o;

always @(*) begin
    case (instr_addr_i)
        0: instr_o = 14'b000000100110011;
        1: instr_o = 14'b000000101010100;
        2: instr_o = 14'b000000101110101;
        3: instr_o = 14'b000000110010110;
        4: instr_o = 14'b000000000010011;
        5: instr_o = 14'b000000000110100;

```

6: instr_o = 14'b000000001010101;
7: instr_o = 14'b000000001110110;
8: instr_o = 14'b000010000000011;
9: instr_o = 14'b000010000100011;
10: instr_o = 14'b000010001000011;
11: instr_o = 14'b000010001100011;
12: instr_o = 14'b00100000001001;
13: instr_o = 14'b00100000101010;
14: instr_o = 14'b00100001001011;
15: instr_o = 14'b00100001101100;
16: instr_o = 14'b00110000010111;
17: instr_o = 14'b00110001101000;
18: instr_o = 14'b00110001000111;
19: instr_o = 14'b00110000100110;
20: instr_o = 14'b00110000000101;
21: instr_o = 14'b00000100101101;
22: instr_o = 14'b00100100110001;
23: instr_o = 14'b01000111001001;
24: instr_o = 14'b00100110101111;
25: instr_o = 14'b00110011001101;
26: instr_o = 14'b01010100100000;
27: instr_o = 14'b00000100100100;
28: instr_o = 14'b00010100100011;
29: instr_o = 14'b00010100000011;
30: instr_o = 14'b00010011100011;
31: instr_o = 14'b00010011000011;
32: instr_o = 14'b00010010100011;
33: instr_o = 14'b00010010000011;
34: instr_o = 14'b00000100100100;
35: instr_o = 14'b00000010000101;
36: instr_o = 14'b00000010100110;
37: instr_o = 14'b00000011000111;
38: instr_o = 14'b00000011101000;
39: instr_o = 14'b00000100001001;
40: instr_o = 14'b01100100000000;
41: instr_o = 14'b01100001100001;
42: instr_o = 14'b01100001000001;
43: instr_o = 14'b01100000100001;
44: instr_o = 14'b01100000000001;
45: instr_o = 14'b01010100100000;
46: instr_o = 14'b01110000100000;
47: instr_o = 14'b01110001100010;
48: instr_o = 14'b01010110000001;
49: instr_o = 14'b10000000100000;
50: instr_o = 14'b10000001100010;
51: instr_o = 14'b01010101100001;
52: instr_o = 14'b10010000100000;
53: instr_o = 14'b10010001100010;
54: instr_o = 14'b01010101000001;
55: instr_o = 14'b10100000100000;

```

56: instr_o = 14'b10100001100010;
57: instr_o = 14'b01010100100001;
58: instr_o = 14'b000000000001001;
59: instr_o = 14'b00000000101010;
60: instr_o = 14'b00000001001011;
61: instr_o = 14'b00000001101100;
62: instr_o = 14'b01001000010010;
63: instr_o = 14'b01000111010010;
default: instr_o = 0;
endcase
end
endmodule

```

data_mem.v:

```

`timescale 1ns / 1ps
module data_mem(clk, rst, data_addr_i, data_o, data_i, data_wr_en, s1_i, s2_i,
s3_i, s4_i, ct_o);

parameter DATA_ADDR_SIZE = 5;
parameter DATA_SIZE = 16;
parameter RAM_DEPTH = 24;

input clk, rst, data_wr_en;
input [DATA_ADDR_SIZE-1:0] data_addr_i;
input [DATA_SIZE-1:0] data_i;
output reg [DATA_SIZE-1:0] data_o;
input [15:0] s1_i, s2_i, s3_i, s4_i;
output [63:0] ct_o;

reg [DATA_SIZE-1:0] memory[0:RAM_DEPTH-1];
reg [DATA_SIZE-1:0] memoryNext[0:RAM_DEPTH-1];

assign ct_o = {memory[3], memory[2], memory[1], memory[0]};

always @(posedge clk) begin
    data_o <= #1 memory[data_addr_i];

    memory[0] <= #1 memoryNext[0];
    memory[1] <= #1 memoryNext[1];
    memory[2] <= #1 memoryNext[2];
    memory[3] <= #1 memoryNext[3];
    memory[4] <= #1 memoryNext[4];
    memory[5] <= #1 memoryNext[5];
    memory[6] <= #1 memoryNext[6];
    memory[7] <= #1 memoryNext[7];
    memory[8] <= #1 memoryNext[8];
    memory[9] <= #1 memoryNext[9];

```

```

memory[10] <= #1 memoryNext[10];
memory[11] <= #1 memoryNext[11];
memory[12] <= #1 memoryNext[12];
memory[13] <= #1 memoryNext[13];

end

always @(*) begin
    memory[14] = 63;
    memory[15] = 4;
    memory[16] = 17;
    memory[17] = 65412;
    memory[18] = 0;
    memory[19] = s1_i;
    memory[20] = s2_i;
    memory[21] = s3_i;
    memory[22] = s4_i;
    memory[23] = 65535;
end

always @(*) begin
    if(rst) begin
        memoryNext[0] = 16'b0;
        memoryNext[1] = 16'b0;
        memoryNext[2] = 16'b0;
        memoryNext[3] = 16'b0;
        memoryNext[4] = 0;
        memoryNext[5] = 1;
        memoryNext[6] = 2;
        memoryNext[7] = 3;
        memoryNext[8] = 4;
        memoryNext[9] = 16'b0;
        memoryNext[10] = 16'b0;
        memoryNext[11] = 16'b0;
        memoryNext[12] = 16'b0;
        memoryNext[13] = 16'b0;
    end else begin
        memoryNext[0] = memory[0];
        memoryNext[1] = memory[1];
        memoryNext[2] = memory[2];
        memoryNext[3] = memory[3];
        memoryNext[4] = memory[4];
        memoryNext[5] = memory[5];
        memoryNext[6] = memory[6];
        memoryNext[7] = memory[7];
        memoryNext[8] = memory[8];
        memoryNext[9] = memory[9];
        memoryNext[10] = memory[10];
        memoryNext[11] = memory[11];
    end
end

```

```

        memoryNext[12] = memory[12];
        memoryNext[13] = memory[13];

        if(data_wr_en) begin
            memoryNext[data_addr_i] = data_i;
        end
    end

end

endmodule

```

present.asm:

```

// pre-processing
// copy previous sensor values
0: CP 209 219
1: CP 210 220
2: CP 211 221
3: CP 212 222

// copy next sensor values
4: CP 200 219
5: CP 201 220
6: CP 202 221
7: CP 203 222

// divide next sensor values by 8
8: SRLi 200 3
9: SRLi 201 3
10: SRLi 202 3
11: SRLi 203 3

// sum sensor values
12: ADD 200 209
13: ADD 201 210
14: ADD 202 211
15: ADD 203 212

// reverse lsb 16 bit of the sum
16: XOR 200 223

// state = state xor rk
17: XOR 203 208
18: XOR 202 207
19: XOR 201 206
20: XOR 200 205

// jump halt if (counter == 32*4)

```

```

21: CP 209 213
22: ADD 209 217
23: BZJ 214 209

// key schedule start
// key = key xor round_counter
24: ADD 213 215
25: XOR 206 213

// key = key r>> 3
26: GETW 209 0 // zeroize W
27: CP 209 204 // save key_0
28: SRLi 209 3
29: SRLi 208 3
30: SRLi 207 3
31: SRLi 206 3
32: SRLi 205 3
33: SRLi 204 3

// key = key r>> 16
34: CP 209 204
35: CP 204 205
36: CP 205 206
37: CP 206 207
38: CP 207 208
39: CP 208 209

// key = SBOX(key[79:76]) & key
40: SBOX 208 0
// key schedule end

// encrypt - SBox
41: SBOX 203 1
42: SBOX 202 1
43: SBOX 201 1
44: SBOX 200 1

// encrypt - PLayer
45: GETW 209 0
46: PER3 201 200
47: PER3 203 202
48: GETW 212 1
49: PER2 201 200
50: PER2 203 202
51: GETW 211 1
52: PER1 201 200
53: PER1 203 202
54: GETW 210 1
55: PER0 201 200
56: PER0 203 202

```

```

57: GETW 209 1
58: CP 200 209
59: CP 201 210
60: CP 202 211
61: CP 203 212

// jump to the loop beginning for 32 rounds
62: BZJ 216 218

// Goto $HALT -- Instruction jumps to itself to terminate the process.
63: BZJ 214 218

////////////////////////////////////
////////////////////////////////////

// Data registers
// state = msb[3 2 1 0]lsb
200: 0 // state_0
201: 0 // state_1
202: 0 // state_2
203: 0 // state_3
// key = msb[4 3 2 1 0]lsb
204: 0 // key_0
205: 1 // key_1
206: 2 // key_2
207: 3 // key_3
208: 4 // key_4
// temporary registers
209: 0 // temp_0
210: 0 // temp_1
211: 0 // temp_2
212: 0 // temp_3
// other values
213: 0 // counter
214: 63 // halt address
215: 4 // 4
216: 17 // loop beginning
217: 65412 //  $2^{16}-31*4$ 
218: 0 // Zero
// sensor inputs
219: 11111 // s1_i
220: 22222 // s2_i
221: 33333 // s3_i
222: 44444 // s4_i
223: 65535 // ffff

```

BIBLIOGRAPHY

- [1] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer Science & Business Media, 2009.
- [2] SSL2BUY, *Symmetric vs. Asymmetric Encryption – What are differences?*, Retrieved from <https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences>, 4 August 2020.
- [3] D. D. Dinu, *Efficient and Secure Implementations of Lightweight Symmetric Cryptographic Primitives*, Doctoral Dissertation, Luxemburg University, Luxemburg, 2017.
- [4] Siber Vatan, *Scytale Şifreleme*, Retrieved from <https://www.sibervatan.org/makale/scytale-sifreleme/19>, 4 August 2020.
- [5] A. Varici, G. Saglam, S. Ipek, A. Yildiz, S. Gören, A. Aysu, D. Iskender, T. B. Aktemur, and H. F. Ugurdag, *Fast and Efficient Implementation of Lightweight Crypto Algorithm PRESENT on FPGA through Processor Instruction Set Extension*, in Proc. IEEE East-West Design & Test Symposium (EWDTS), pp. 43-47, 2019.
- [6] Tasarım Kodlama. *Internet of Things – Nesnelerin İnterneti*, Retrieved from <https://www.tasarimkodlama.com/genel/teknoloji/internet-of-things-nesnelerin-interneti/>, 4 August 2020.
- [7] J. Daemen and V. Rijmen, *AES proposal: Rijndael*, 1999.
- [8] D. E. Standard, *Federal Information Processing Standards Publication 46*, National Bureau of Standards, US Department of Commerce, 1977.
- [9] K. McKay, L. Bassham, M. S. Turan, and N. Mouha, *Report on Lightweight Cryptography*, NISTIR 8114 (Draft), National Institute of Standards and Technology, 2016.
- [10] A. Yıldız, H. F. Ugurdag, B. Aktemur, D. İskender, and S. Gören, *CPU design simplified*, in Proc. IEEE International Conference on Computer Science and Engineering (UBMK), pp. 630-632, 2018.
- [11] A. Poschmann, *Lightweight Cryptography-Cryptographic Engineering for a Pervasive World*, Doctoral Dissertation, Ruhr University Bochum, Germany, 2009.
- [12] Sparkfun, *Integrated Circuits*, Retrieved from <https://learn.sparkfun.com/tutorials/integrated-circuits/all>, 4 August 2020.
- [13] Opengenius, *Structure of Field Programmable Gate Array (FPGA)*, Retrieved from <https://iq.opengenus.org/structure-of-field-programmable-gate-array-fpga/>, 4 August 2020.
- [14] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. S. Koo, D. Chang, J. Lee, K. Jeong, H. Kim, and S. Chee, *HIGHT: A New Block Cipher Suitable for Low-resource Device*, in Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp. 46-59, Springer, Berlin, Heidelberg, 2006.

- [15] C. H. Lim and T. Korkishko, *mCrypton – A Lightweight Block Cipher for Security of Low-cost RFID Tags and Sensors*, in Proc. International Workshop on Information Security Applications (WISA), pp. 243-258, Springer, Berlin, Heidelberg, 2005.
- [16] F. X. Standaert, G. Piret, N. Gershenfeld, and J. J. Quisquater, *SEA: A Scalable Encryption Algorithm for Small Embedded Applications*, in Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS), pp. 222-236, Springer, Berlin, Heidelberg, 2006.
- [17] D. J. Wheeler and R. M. Needham, *TEA, A Tiny Encryption Algorithm*, in Proc. International Workshop on Fast Software Encryption (FSE), pp. 363-366, Springer, Berlin, Heidelberg, 1994.
- [18] F. X. Standaert, G. Piret, G. Rouvroy, J. J. Quisquater, and J. D. Legat, *ICEBERG: An Involutional Cipher Efficient for Block Encryption in Reconfigurable Hardware*, in Proc. International Workshop on Fast Software Encryption (FSE), pp. 279-298, Springer, Berlin, Heidelberg, 2004.
- [19] Github, *Present-vhdl*, Retrieved from <https://github.com/huljar/present-vhdl>, 4 August 2020.
- [20] A. Johansson and J. Johansson, *High-Level Synthesis for Efficient Design and Verification*, Master's Thesis, Lund University, Lund, 2020.
- [21] D. Dinu, A. Biryukov, J. Großschädl, D. Khovratovich, Y. Le Corre, and L. Perrin, *Felics – Fair Evaluation of Lightweight Cryptographic Systems*, in Proc. NIST Workshop on Lightweight Cryptography, vol. 128, 2015.
- [22] Xilinx, *PicoBlaze 8-bit Embedded Microcontroller User Guide (UG129)*, 2011.
- [23] E. B. Kavun and T. Yalcin, *RAM-based Ultra-lightweight FPGA Implementation of PRESENT*, in Proc. IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 280-285, 2011.
- [24] J. J. Tay, M. D. Wong, M. M. Wong, C. Zhang, and I. Hijazin, *Compact FPGA Implementation of PRESENT with Boolean S-Box*, in Proc. IEEE Asia Symposium on Quality Electronic Design (ASQED), 2015.
- [25] C. A. Lara-Nino, M. Morales-Sandoval, and A. Diaz-Perez, *Novel FPGA-based Low-cost Hardware Architecture for the PRESENT Block Cipher*, in Proc. IEEE Euromicro Conference on Digital System Design (DSD), pp. 646-650, 2016.
- [26] W. Diehl, F. Farahmand, P. Yalla, J. P. Kaps, and K. Gaj, *Comparison of Hardware and Software Implementations of Selected Lightweight Block Ciphers*, in Proc. IEEE International Conference on Field Programmable Logic and Applications (FPL), pp. 1-4, 2017.
- [27] S. Engels, E. B. Kavun, C. Paar, T. Yalcin, and H. Mihajloska, *A Non-linear/Linear Instruction Set Extension for Lightweight Ciphers*, in Proc. IEEE Symposium on Computer Arithmetic (ARITH), pp. 67-75, 2013.

VITA

Abdullah Varıcı completed his Bachelor's in Electronics and Communication Engineering from Istanbul Technical University in 2017. After graduating, he started his master's education in Electrical and Electronics Engineering at Özyeğin University in 2018. While studying there, he became a member of the “nEMESysLab” at Özyeğin University and he worked on developing processors for lightweight cryptography with Professor H. Fatih Uğurdağ. He has also been working at Procenne as a digital design engineer since 2017, where he develops high-performance cryptographic cores and performs side-channel analysis on them.