THE FAST AND EFFICIENT IMPLEMENTATION OF PRINCE ALGORITHM ON FPGAS BY EXTENDING THE INSTRUCTION SET OF A PROCESSOR

by
Tugay Emre Yücedağ

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2021

THE FAST AND EFFICIENT IMPLEMENTATION OF PRINCE ALGORITHM ON FPGAS BY EXTENDING THE INSTRUCTION SET OF A PROCESSOR

APPROVED BY:

Prof. Dr. Sezer Gören Uğurdağ ……
(Thesis Supervisor)
(Yeditepe University)

Prof. Dr. Hasan Fatih Uğurdağ ……
(Thesis Co-Supervisor)
(Özyeğin University)

Prof. Dr. Gürhan Küçük ……
(Yeditepe University)

Assist. Prof. Dr. Vecdi Emre Levent ……
(Fenerbahçe University)

DATE OF APPROVAL: ..../..../2021

I hereby declare that this thesis is my own work and that all information in this thesis has been obtained and presented in accordance with academic rules and ethical conduct. I have fully cited and referenced all material and results as required by these rules and conduct, and this thesis study does not contain any plagiarism. If any material used in the thesis requires copyright, the necessary permissions have been obtained. No material from this thesis has been used for the award of another degree.

I accept all kinds of legal liability that may arise in case contrary to these situations.

Name, Last name ……

Signature …..…….……..

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my advisor Prof. Dr. Sezer Gören Uğurdağ and co-advisor Prof. Dr. Hasan Fatih Uğurdağ in my project for their direction, trust, and support.

In addition, I would like to thanks to Dr. Abdullah Yıldız, for helping me in every condition for my thesis.

I would like to share my appreciation to TÜBİTAK for including me as a scholarship student of TÜBİTAK 1001 “Multi-Core Microcontroller Synthesis from C” project (no: 117E090). In this study, several outcomes of this TÜBİTAK project were used to develop the conclusion.

Finally, I would like to thanks to my dear wife Elif Er Yücedağ, my dear mom Sevda Yücedağ, my father Bülent Yücedağ, Dur family, Er family, my dear grandmother Perihan Özdil and my whole family, for their support and love.

ABSTRACT

THE FAST AND EFFICIENT IMPLEMENTATION OF PRINCE ALGORITHM ON FPGAS BY EXTENDING THE INSTRUCTION SET OF A PROCESSOR

In this study, the Prince algorithm, which is one of the cryptographic algorithms developed to be used for IoT devices, has been implemented in an efficient way in terms of latency and resource usage criteria. The mentioned criteria are significant in almost all of the IoT applications that are used recently. In order to ensure success in the mentioned criteria, the algorithm has been implemented with Pure Register Transfer Level (RTL) design, High Level Synthesis (HLS), standard processors, and customizable processor designs.

The main object of this study is to show that the customizable processor is available to meet the criteria mentioned above when the necessary changes have occurred. As a result of the studies, it has been determined that the customizable processor works more optimized than other implementations in terms of latency and resource usage balance, along with the design changes.

ÖZET

PRINCE ALGORİTMASI'NIN BİR İŞLEMCİNİN KOMUT KÜMESİ GENİŞLETİLMESİYLE FPGA'LER ÜZERİNE HIZLI VE VERİMLİ GERÇEKLEMESİ

Bu calışmada, IoT cihazları için geliştirilen kriptografik algoritmalardan biri olan Prince algoritması gecikme ve kaynak kullanımı kriterleri açısından verimli olacak bir şekilde gerçeklenmiştir. Bahsedilen kriterler günümüz IoT uygulamalarının hemen hemen hepsinde önem arz etmektedir. Bahsedilen kriterlerde başarıyı sağlamak için algoritma, Saf Kaydedici Transfer Seviyesi (RTL) tasarımı, Yüksek Seviye Sentez (HLS), standart işlemciler ve özelleştirilebilen işlemci tasarımları ile gerçeklenmiştir.

Bizim bu calışmada asıl amacımız; özelleştirilebilir işlemcinin gerekli değişiklikleri yapıldığı taktirde yukarıda bahsedilen kriterleri sağlayabileceğini göstermektir. Yapılan çalışmalar sonucunda da tasarımsal değişiklikler ile birlikte özelleştirilebilir işlemcinin gecikme ve kaynak kullanımı dengesi bakımından diğer gerçeklemelerden daha optimize bir şekilde çalıştığı saptanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS / ABBREVIATIONS

ALU	Arithmetic logic unit
ARM	Advanced RISC machines microcontroller
ASIC	Application specific integrated circuits
AVR	Advanced virtual RISC microcontroller
CLB	Configurable logic blocks
CPU	Central processing unit
FELICS	Fair evaluation of lightweight cryptographic systems
FPGA	Field programming gate arrays
HDL	Hardware description language
I/O	Input / Output
IC	Integrated circuit
IEEE	The Institute of Electrical and Electronics Engineers
IoT	Internet of things
ISA	Instruction set architecture
LUT	Look-up table
MSP	Mixed-signal microcontroller
RAM	Random access memory
RC	Round-dependent constant
ROM	Read only memory
SPN	Substitution-permutation network
VSCPU	Very simple central processing unit
XOR	Exclusive OR
VSCPU-32-P-C	Customized 32-bit VSCPU
VSCPU-64-NP-H	Non-pipelined 64-bit VSCPU with Harvard architecture
VSCPU-64-NP-V	Non-pipelined 64-bit VSCPU with Von Neumann architecture
VSCPU-64-P	Pipelined 64-bit VSCPU

1. INTRODUCTION

Knowledge has been the most important phenomenon in the world throughout human history. This importance is constantly increasing day by day with the developments in technology. Not all knowledge is open to the public, some only to be understood by those authorized to learn it. This can sometimes be the location of a treasure, sometimes the coordinates of the extraction to be made in a war. In order to provide this confidentiality, humanity discovered cryptography thousands of years ago.

Cryptology [1] is a branch of science concerned with safe data transmission and storage. Cryptography and cryptanalysis are both included in cryptology. Cryptography is the study of secret writing with the objective of concealing a message's meaning. Cryptanalysis is the science and practice of decrypting cryptographic systems in an unauthorized way.

Information or messages that are tried to be hidden are called plaintext. In order to securely transport and understand these messages or information, keys are used which are the information that is known by the authorized people. In cryptography, there are several different types of keys. There are two processes for transporting information securely. The first one is encryption. Encryption is the process of converting data (plaintext) into a secret code (ciphertext) with a key that hides the true meaning of the data. The second one, which is called decryption, is the process of restoring encrypted data to its original state.

Cryptography is divided into three branches [2]: symmetric and asymmetric cryptography, cryptographic protocols. The old one is symmetric cryptography. From ancient times until 1976, all cryptography relied solely on symmetric methods [2]. The same keys are used for encryption and decryption.

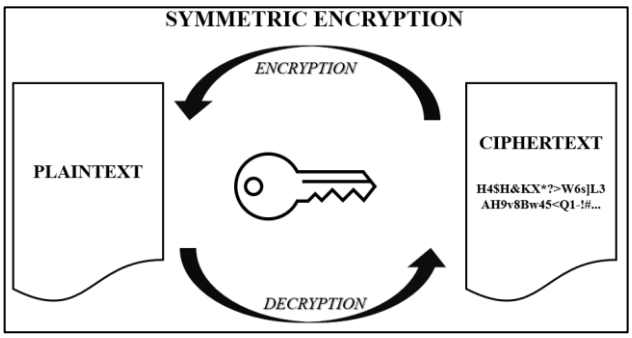


Figure 1.1. Symmetric encryption and decryption

Asymmetric encryption was developed in response to the necessity for a novel approach to encryption in today's digital environment. A pair of linked keys are employed in this strategy, which consists of a public key for encrypting data and a private key for decrypting data.

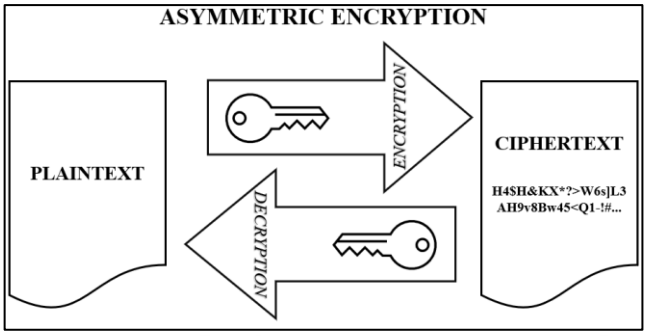


Figure 1.2. Asymmetric encryption and decryption

Both methods of cryptography have advantages and weaknesses, and they can be employed in a variety of cryptographic protocols together or independently. Symmetric and asymmetric algorithms [2] can be thought of as building blocks that can be used to create applications like secure Internet communication.

Symmetric ciphers are separated into stream ciphers and block ciphers. Stream ciphers encryption is performed bit by bit but block ciphers use the same key to encrypt an entire block, which is obtained by dividing the whole into parts, of plaintext bits at once. Some popular block ciphers are AES (Advanced Encryption Standard), DES (Data Encryption Standard), IDEA (International Data Encryption Algorithm), Blowfish, RC5, and RC6.

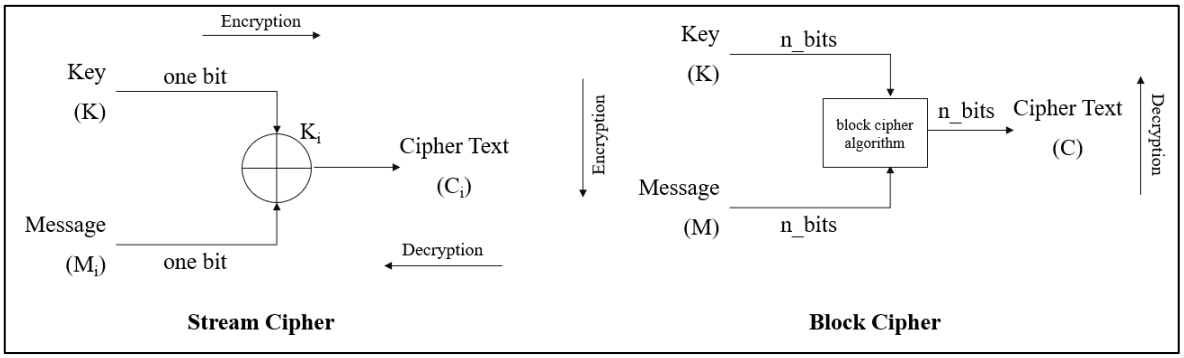


Figure 1.3. Stream and Block Cipher

Cryptography has been continuously developed and diversified. With the development of internet technologies, the importance of cryptography is increasing.

Humanity lives in an era called smart. What is the meaning of smart? For example, a smartwatch is a device that screens the sleep quality along with the general health condition information of the person who wears that. Also, a smartphone could be a person's assistant to operate a coffee machine while he or she is still sleeping. Meanwhile, all these and similar things are happening, the smart devices are actually communicating with each other over the internet. This is called the Internet of Things.

The Internet of Things (IoT) [3] is a network of physical objects (or "things") that are implanted with sensors, software, and other technologies in order to connect and exchange data with other devices and systems over the internet. As can be understood from this definition, there is a growing problem that is the security of information between IoT devices.

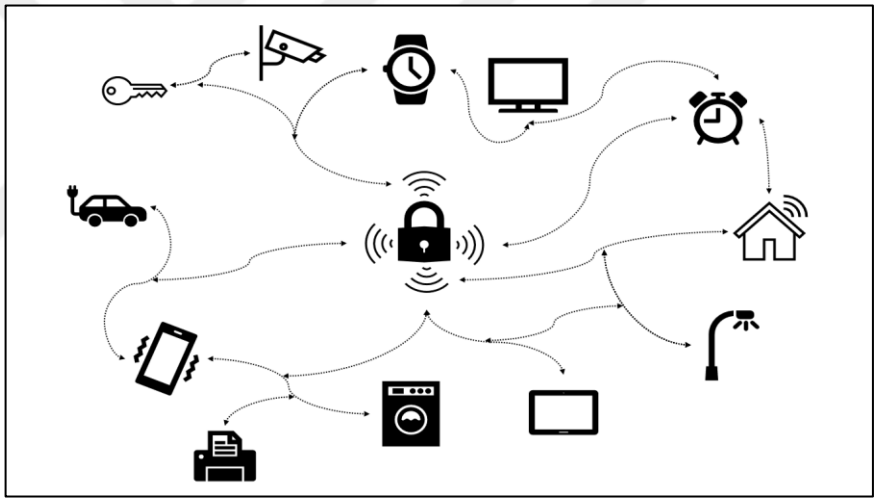


Figure 1.4. Security and IoT

1.1. MOTIVATION

Houses, cars, watches, and even glasses are becoming connected to the internet day by day. It is desired by people to remotely control every device and observe themselves any time they want. There are now new smart devices for all of these like smart scales, smart thermos, smartwatches, etc. To sum up, every breath we take is the data to be transmitted between our devices.

However, this data does not always go to the cloud from a desktop computer or smartphone. It often goes from a very small smartwatch to an air purifier whose main job is not

computing. That is why IoT devices often do not have a lot of computational resources, considering both price and their core business.

While this much connectivity is great, it comes with a lot of security issues. This is where Lightweight cryptography comes into play, and it provides security with limited resources.

At this point, it has been researched how we can overcome these constraints by using a customizable processor.

1.2. CONTRIBUTION

As mentioned before, limited resources are the main constraints of IoT devices. In this study, we focus PRINCE algorithm, which is developed to provide security within these limits.

The contribution of this study to the literature is the implementation of PRINCE in several ways mentioned below:

- Implementation with Pure-RTL
- Implementation with Vivado HLS
- Implementation with Standart Processor
- Implementation with Customizable Processor

Obtained results compared with the results in the literature in terms of resource usage and timing.

2. BACKGROUND

This chapter will cover some background knowledge on lightweight cryptography, CPUs for microcontroller applications that commonly use IoT devices, basic CPU architectures, FPGA, and evaluation metrics for digital design.

2.1. LIGHTWEIGHT CRYPTOGRAPHY

Considering the encryption technologies, there are many encryption algorithms with proven reliability such as AES [4], DES [5], etc. However, these encryption technologies have been developed to be used in areas such as desktops, servers, etc., where there will be no shortage of resources in terms of energy, space, power, etc. Therefore, it is not possible to use the mentioned old-style encryption algorithms in applications with limited resources such as IoT. Because of this problem, new encryption algorithms are needed. Lightweight encryption algorithms have emerged to meet this need. This study focuses on block cipher lightweight cryptography. Block cipher lightweight cryptography algorithms differ from each other in terms of structure [6] which are Substitution-Permutation Network (SPN), Feistel Network (FN), General Feistel Network (GFN), Add-Rotate-XOR (ARX), Nonlinear-Feedback Shift Register (NLFSR), and Hybrid.

Table 2.1. Structure of block cipher lightweight cryptography algorithms [6]

Structure	Algorithm
SPN	AES, Present, GIFT, SKINNY, Rectangle, Midori, mCrypton, Noekon, Iceberg, Puffin-2, Prince, Pride, Print, Klein, Led, Picaro, Zorro, I-Present, EPCBC, etc.
FN	DESL/DESXL, TEA/XTEA/XXTEA, Camelia, Simon, SEA, KASUMI, MIBS, Lblock, ITUbee, FeW, GOST, Robin, Fantomas, etc.
GFN	CLEFIA, Piccolo, Twis, Twine, HISEC, etc.
ARX	SPECK, IDEA, HIGHT, BEST-1, LEA, etc.
NLFSR	KeeLoq, KATAN/KTANTAN, Halka, etc.
Hybrid	Hummingbird, Hummingbird-2, Presen-GRP, etc.

SPN and FN are the structures that stand out among these structures because they can be adapted according to the requirements of the application. The substitution-permutation network (SPN) modifies the data using a series of substitution boxes and permutation tables, preparing it for the next round. The block is split into two equal halves in Feistel Network (FN). These halves are called the right and left half. Every round, right half remains same but left round is changing by a series operation. At the beginning of each round, left and right halves are swapped, thus the whole block is encrypted.

2.2. FIELD PROGRAMMABLE GATE ARRAY

FPGAs [7] (Field Programmable Gate Arrays) are semiconductor devices that consist of a matrix of customizable logic blocks (CLBs) linked by programmable interconnects which allows a developer to create digital circuits as desired. On contrary to ASIC (Application Specific Integrated Circuit), FPGA is a programmable IC. Thanks to its programmability feature, we can program it as a microprocessor or do a more specific job.

A CLB [8] is the heart of an FPGA, and it's what allows it to work with a variety of hardware combinations. In fact, FPGA is a pile that holds a bunch of CBL. Look-Up Tables (LUTs), Multiplexers, and Flip-Flops are the most common components of CLBs.

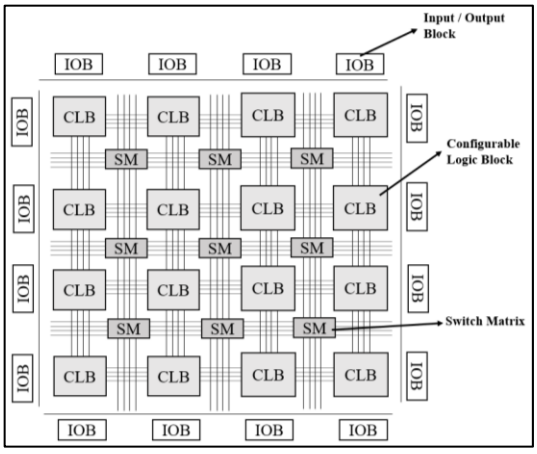


Figure 2.1. Structure of FPGA

In this study, all implementations will be implemented on FPGA.

2.3. HLS

Another way to design RTL is using HLS (High Level Synthesis). HLS [9] makes it possible to develop a synthesizable RTL design with high-level languages like C, C++, System C. It enables designers to work at the level of high-level software languages, decreasing development time and optimizing performance. In this study, Xilinx Vivado HLS is used.

2.4. CPU ARCHITECTURE

The most common CPU architectures are Von Neumann and Harvard architecture. Von Neumann architecture is designed by John Von Neumann in 1945. Basically, in Von Neumann architecture, both data and instruction are kept in the same memory block. For this reason, the CPU cannot read instructions while it reads or writes data. The second one is Harvard architecture. The starting point of developing Harvard architecture is Von Neumann bottleneck. Because of using the same bus to access program and data memory, bottlenecks have occurred. That is why in Harvard architecture data and instruction are kept in separate memory blocks to prevent bottlenecks.

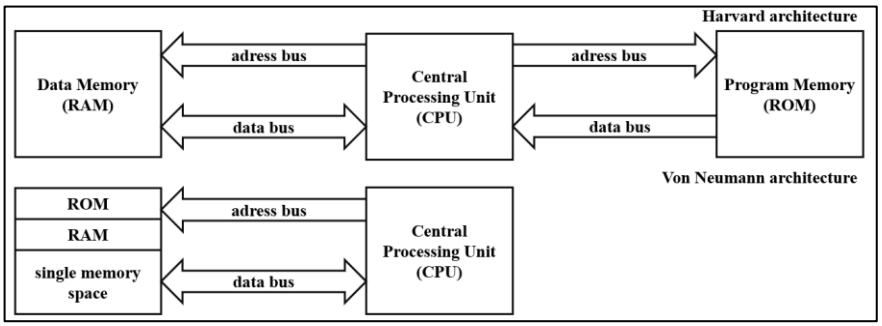


Figure 2.2. Harvard and Von Neumann Architecture

2.5. PICOBLAZE

PicoBlaze [10] is an 8-bit microcontroller with Reduced Instruction Set Computing (RISC) which is optimized for Xilinx FPGAs in terms of area. Unlike its area consumption, it comes with low performance. PicoBlaze has 70 instructions. Each instruction is executed in two cycles.

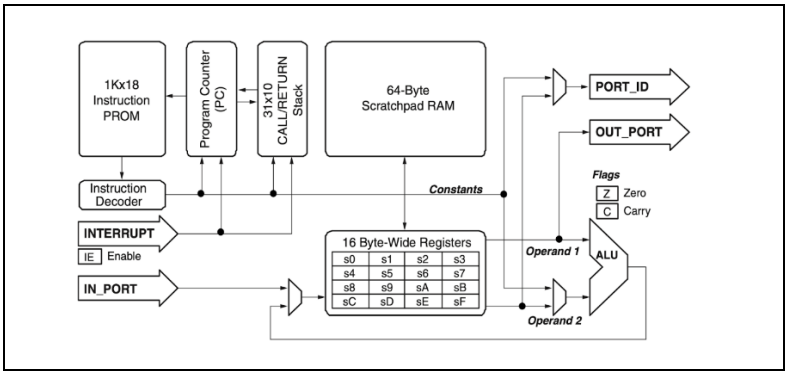


Figure 2.3. Block diagram of PicoBlaze [10]

The PicoBlaze's block diagram in Figure 5.1 depicts the major functional blocks and I/O. Some of PicoBlaze features mentioned below [10]:

- Data register for general-purpose 16-byte
- Program memory for storing 1K instruction
- ALU includes CARRY and Zero flags 8-bit wide
- Internal scratchpad RAM 64 byte
- For easy expansion and enhancement, there are 256 input and output ports
- Automatic 31-location CALL/RETURN stack
- Interrupt support
- Instruction-set simulator support and Assembler

2.6. VSCPU

VSCPU stands for Very Simple CPU, and it is the main processor in this study. Customizable and simple structure of the processor helps in the fast and efficient implementation of the PRINCE algorithm. It is utilized in this part without any changes.

Besides, the VSCPU [11] is a basic and customizable 32-bit processor, it has some tools that are simulator for instruction set, assembler, and a special C compiler for converting C code into VSCPU assembly code. There are two main aims to develop VSCPU. The first one is

to generate an application-specific multi-core microcontroller using a program that is written in C language. To achieve this purpose, the special C compiler generates VSCPU assembly code. Then the instruction set simulator generates a memory file for VSCPU processor. The second aim is to provide computer science students to understand of how CPU works, CPU architecture and develop VSCPU with RTL code and write programs for VSCPU using the tools which mentioned above.

VSCPU has a fixed length of 32-bit instruction word, its instruction set comprises 16 instructions and supports unsigned integer arithmetic. Verilog language was used to develop RTL code and its design could be fully modified if necessary. VSCPU supports interrupt and it can be used to access a variety of peripherals and interfaces, including GPIO, RS232, VGA, I2C, SPI, and others. In the VSCPU, there are four registers:

- PC: 14-bit wide Program Counter, it holds information of where program left
- IW: 32-bit wide Instruction Word, it holds which instruction will be executed
- R1: 32-bit wide General Purpose Register, it keeps data or address
- R2: 32-bit wide General Purpose Register, it keeps data or address like R1 but it is optional

Table 2.2. Instruction word structure of VSCPU [11]

Description	**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

While the origin of VSCPU has 7 states, in this study, the implementation of the PRINCE algorithm does not need interrupt states.

Table 2.3. States of VSCPU [11]

States	Description
Fetch instruction	The instruction fetched from memory which is addressed by PC
Fetch operand A	IW is decoded and operand A value is used as next address to fetch memory
Fetch operand B	Operand B value fetch from memory
Execute instruction and write back	After ALU was performed, the result write back to memory and the PC value changed

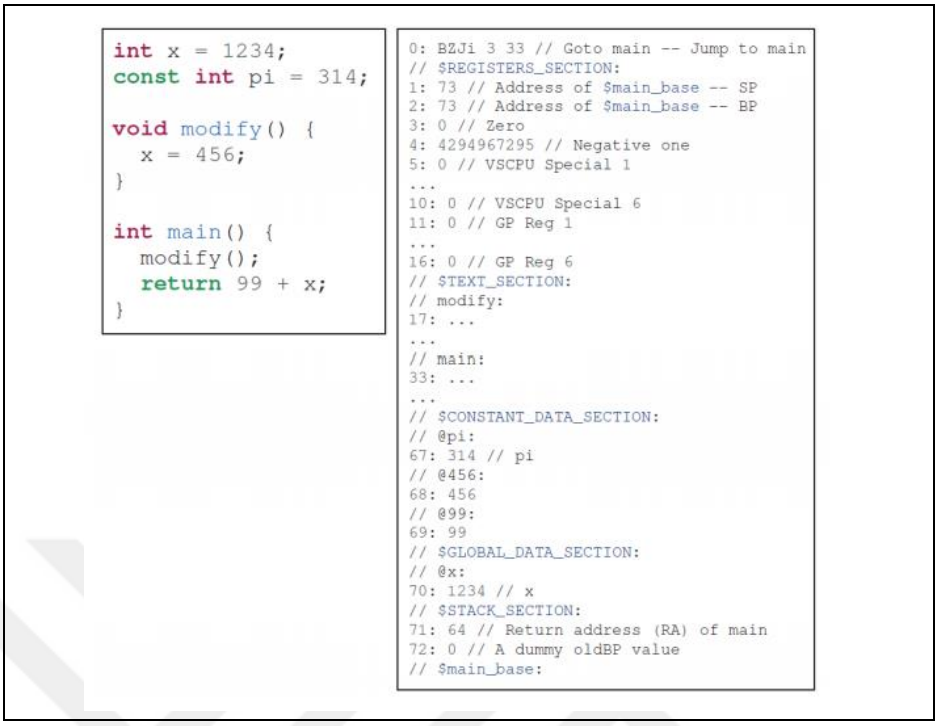
Table 2.4. Instructions of VSCPU [11]

Mnemonic	Description	Microoperations
ADD A B	unsigned Add	mem[A] ← mem[A] + mem[B]
ADDi A B	unsigned Add immediate	mem[A] ← mem[A] + B
NAND A B	bitwise NAND	mem[A] ← ~(mem[A] & mem[B])
NANDi A B	bitwise NAND immediate	mem[A] ← ~(mem[A] & B)
SRL A B	shift right/left	mem[A] ← (mem[B] < 32) ? (mem[A] >> mem[B]) : (mem[A] << (mem[B]-32))
SRLi A B	shift right/left immediate	mem[A] ← (B<32)?(mem[A]>>B):(mem[A]<<(B-32))
LT A B	compare and set	mem[A] ← (mem[A] < mem[B]) ? 1 : 0
LTi A B	compare and set immediate	mem[A] ← (mem[A] < B) ? 1 : 0
MUL A B	unsigned multiply	mem[A] ← mem[A] × mem[B]

Table 2.4 continued

MULi A B	unsigned multiply immediate	mem[A] ← mem[A] × B
CP A B	copy data	mem[A] ← mem[B]
CPi A B	copy data immediate	mem[A] ← B
CPI A B	copy data indirect	mem[A] ← mem[mem[B]]
CPIi A B	copy data indirect immediate	mem[mem[A]] ← mem[B]
BZJ A B	branch on zero	PC ← (mem[B] == 0) ? mem[A] : (PC + 1)
BZJi A B	unconditional branch	PC ← mem[A] + B

The original implementation flow of VSCPU starts with converting C code to compatible format for the VSCPU C Compiler which preprocesses, parses, and semantically analyzes code with Clang. If there is no error in the source code, the compiler generates assembly code, then generated assembly code is simulated with the instruction set simulator which is developed with Python language. Instruction set simulator also provides machine code of assembly code after simulation. After that, the machine code was inserted into the memory of VSCPU and it is ready to be synthesized and implemented.



```
int x = 1234;
const int pi = 314;

void modify() {
  x = 456;
}

int main() {
  modify();
  return 99 + x;
}
```

```
0: BZJi 3 33 // Goto main -- Jump to main
// $REGISTERS_SECTION:
1: 73 // Address of $main_base -- SP
2: 73 // Address of $main_base -- BP
3: 0 // Zero
4: 4294967295 // Negative one
5: 0 // VSCPU Special 1
...
10: 0 // VSCPU Special 6
11: 0 // GP Reg 1
...
16: 0 // GP Reg 6
// $TEXT_SECTION:
// modify:
17: ...
...
// main:
33: ...
...
// $CONSTANT_DATA_SECTION:
// @pi:
67: 314 // pi
// @456:
68: 456
// @99:
69: 99
// $GLOBAL_DATA_SECTION:
// @x:
70: 1234 // x
// $STACK_SECTION:
71: 64 // Return address (RA) of main
72: 0 // A dummy oldBP value
// $main_base:
```

Figure 2.4. Source C code and generated assembly code with C compiler [11]

2.7. EVALUATION METRICS

The following metrics are used to compare the results of implementation.

2.7.1. Area

To implement an algorithm on FPGA LUTs and FFs are used in CLBs. That is why the amount of LUTs and FFs gives information about area requirements.

2.7.2. Latency

During the encryption process, plaintext passes through a series of operations to obtain ciphertext. The time from the beginning to the end of these operations is called latency and it is expressed by the clock cycle.

2.7.3. Code Size

In order to find out the memory demand of implementation, code sizes are significant. When the code size is reduced, memory cost is decreased. That is why comparing code sizes gives information about the complexity and performance of the operation.

3. RELATED WORK

The main goal is to create and implement lightweight cryptographic algorithms, that ensure the security level of resource constraint devices such as IoT devices.

There are two types of implementation for lightweight cryptographic algorithms. The first one is the software implementation which is implementing on a processor. The second one is the hardware implementation which is implementing with RTL design. For the software implementation, the required memory size is the main constraint. On the other hand, for the hardware implementation area, speed and power consumption are considered.

Varici et al. [12] implement PRESENT in several ways. These are hardware implementation with hand-coded RTL and Vivado HLS and software implementation with PicoBlaze, VSCPU, and customized VSCPU. According to the results of implementation, customized VSCPU implementation is the best one in terms of resource consumption and performance.

Dinu et al. [13] implement 19 lightweight block ciphers which are AES, Chaskey, Fantomas, HIGHT, LBlock, LEA, LED, Piccolo, PRESENT, PRIDE, PRINCE, RC5, RECTANGLE, RoadRunneR, Robin, Simon, SPARX, Speck, and TWINE on 8, 16 and 32-bit microcontrollers. According to the implementations of block ciphers and considered constraints, Chaskey is the best one.

Diehl et al. [14] implement both ways; hardware and software in Xilinx Kintex-7 FPGAs. For hardware implementation, RTL design is used, for software implementation 8-bit softcore reconfigurable microprocessor which consists of 30 instructions. The microprocessor uses Harvard architecture. To get better implementation results, the memory of the processor is initialized with the minimum amount of memory and unused instructions removed. For hardware implementation, TWINE has the highest throughput-to-area ratio. For software implementation, AES has the lowest cycle-per-byte count.

In another work, Abbas et al. [15] implement Prince block cipher with hardware implementation. Virtex-4 and Virtex-6 FPGA boards are used. In this study researchers aimed to have low latency and low cost. There are 3 ports for the top module interface which are called PRINCE-Core, 2 for input that are plaintext and key, 1 for output that is ciphertext. In the Prince-Core, the algorithm splits into the components that are XOR, S-box layer, M

layer, M' layer, RC constants, and key schedule. All components were built with LUT instead of a block of RAM. Additionally, because the key schedule consumes large space and spends long time, another component is designed to handle the key schedule. That component just includes wiring for 63 shift right operation. The design encrypts data within one clock cycle.

4. PRINCE BLOCK CIPHER

A Substitution-Permutation network [16], often known as a SP network, is a type of block cipher that is made up of rounds of a sequence of mathematical operations as mentioned before. To generate the ciphertext block, the network takes a block of plaintext and the key as inputs and applies numerous alternating "rounds" or "layers" of substitution boxes (S-boxes) and permutation boxes (P-boxes). PRINCE algorithm [17] is an SPN-based structured lightweight block cipher. PRICE has 64-bit block size and 128-bit key size. The algorithm operates 12 rounds. A key addition, a Sbox-layer, a linear layer, and the addition of a round constant are all included in each PRINCE round. The pre and post-whitening keys are made up of the first half of the 128-bit secret key, while the round functions use the remaining 64-bit of the key directly. To decrypt the ciphertext, the key is initially XORed with a preset value before start decryption, and the same circuit used for encryption can be used. As a result [15], the operating costs of decryption are kept at the minimum level.

The main aim of designing the PRINCE algorithm is to have low latency. Previous approaches are emphasized to have a small hardware footprint. If an unrolled implementation is used, PRINCE encryption can be completed in just one clock cycle. Besides, keeping area requirements to the bare minimum, the ideal s-box is selected and the linear layer's operations are kept at the minimum amount by the designers [15].

4.1. DESIGN AND OPERATION DETAILS OF PRINCE

The top design PRINCE [17] algorithm is a function that takes two parameters that are 64-bit plaintext and 128-bit key and generates one output that is ciphertext. When it is looked beneath the surface of this top design, the key schedule will come across.

The key schedule transforms a 128-bit key into three 64-bit keys. It splits the 128-bit key into two equal parts, which are k0 and k1. Another 64-bit key is generated by using k0, k1 that called k0' which is called as round keys. Equation 4.1 shows the notation of generating round keys. The first key k0 is used for pre-whitening and k0' is used for post-whitening.

$$(\boldsymbol{k0} \parallel \boldsymbol{k1}) \rightarrow (\boldsymbol{k0} \parallel \boldsymbol{k0'} \parallel \boldsymbol{k1}) \coloneqq (\boldsymbol{k0} \parallel (\boldsymbol{k0} \gg \boldsymbol{1}) \oplus (\boldsymbol{k0} \gg \boldsymbol{63}) \parallel \boldsymbol{k1}) \quad (4.1)$$

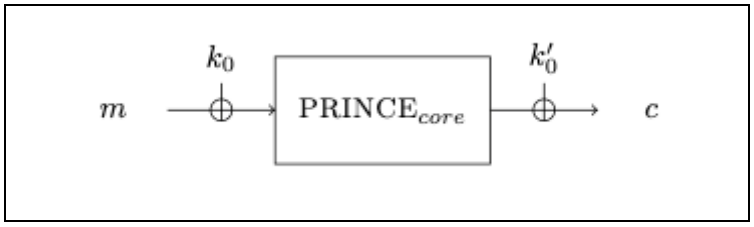


Figure 4.1. Pre and post-whitening [17]

After pre-whitening is performed, the algorithm moves to the core operation that is called PRINCEcore. PRINCEcore's structure is shown in Figure 4.2.

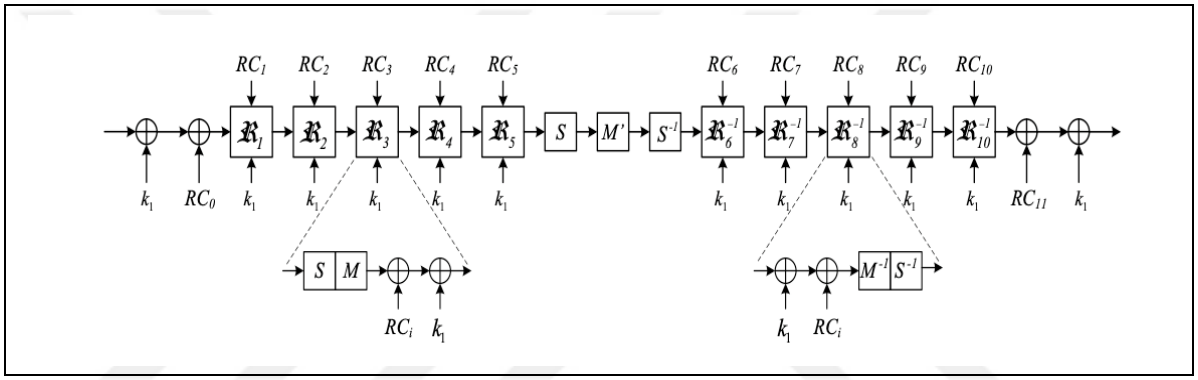


Figure 4.2. Details of PRINCEcore [17]

Every round function includes k1 XORing, RC_i adding, S-layer which is S-box or inverse S-layer, and M-layer or inverse M'-layer.

RC stands for round constant. The PRINCE algorithm has 12 RC [17] constants. $RC_i \oplus RC_{11-i}$ is equal to α for all $0 \leq i \leq 1$, with α = coac29b7c97c50dd (in hexadecimal).

S and Inverse S' layer(S-Box) [15] is used for 4-bit mapping. It takes 4-bit input and maps it to the new value.

Table 4.1. S-Box of PRINCE [15]

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[X]	B	F	3	2	A	C	9	1	6	7	8	0	E	5	D	4
S^{-1}[X]	B	7	3	2	F	D	8	9	A	6	4	0	5	E	C	1

In the linear layer, three input bits are XORed to form a single output. Different input bits are used for each output bit. It is constructed in such a way that diffusion is maximized. [15] Linear layer split into two parts; M-Layer and M'-Layer. M'-layer is just performed for the middle-round and M' has to be involution. The middle involution [17] is made up with the composition of M', SR and SR-1, M = SR ◦ M'. SR, which works in the same way as AES shift rows, permutes the 16 nibbles shown in Figure 4.3. [15]

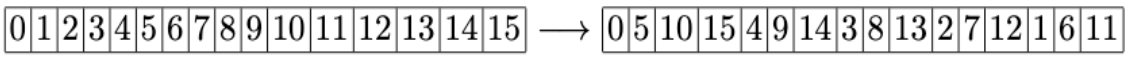


Figure 4.3. SR permutation [17]

Equations 4.2 and 4.3 shows building blocks of M'-layer and M'-layer Diagonal Matrix Elements.

$$M_0=\begin{pmatrix}0&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}\quad M_1=\begin{pmatrix}1&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}$$

$$M_2=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&1\end{pmatrix}\quad M_3=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&0\end{pmatrix} \tag{4.2}$$

$$\widehat{M}^{(0)}=\begin{pmatrix}M_0&M_1&M_2&M_3\\M_1&M_2&M_3&M_0\\M_2&M_3&M_0&M_1\\M_3&M_0&M_1&M_2\end{pmatrix}\qquad \widehat{M}^{(1)}=\begin{pmatrix}M_1&M_2&M_3&M_0\\M_2&M_3&M_0&M_1\\M_3&M_0&M_1&M_2\\M_0&M_1&M_2&M_3\end{pmatrix} \tag{4.3}$$

5. IMPLEMENTATIONS

Prince block cipher algorithm is mentioned above. In this chapter, Prince algorithm is implemented in several ways which are Pure-RTL, Vivado HLS, PicoBlaze, original VSCPU, and customized VSCPU. Obtained results are compared in terms of resource usage and operational time. Since FPGAs are programmable, XC7A200T_ FBG484_-3 FPGA of Xilinx will be used in this study.

5.1. IMPLEMENTATION WITH PURE-RTL

Register-transfer level (RTL) is a design abstraction for modeling a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers and the logical operations performed on those signals in digital circuit design.

The aim is to determine the hardware resource usage and performance of the PRINCE algorithm by hand-coded RTL, before moving on to other implementations. To achieve this aim, RTL design was reviewed which is shared on GitHub [18].

When RTL code is designed, PRINCE algorithm round loops are unrolled. The design is wrapped by a finite state machine that has 4 states. The first state is idle. When the valid signal is active high, it starts to read plaintext, then reads keys and moves to write ciphertext state.

After key reading, in the top-level module, k0’ key is generated and key whitening is performed. Next, in the prince_core module [18] it performs ciphering via all components for 12 rounds. The components are s_box, s_box_inv, linear_m, linear_m_inv, linear_mprime. When the twelfth round is performed, in the prince_top plain text is XORed with k0’ and the ciphertext is written as output.

After implementation, the design used 1880 LUTs and 241 FFs. It can also operate at a clock frequency of up to 68 MHz and calculate results in 20 clock cycles.

5.2. IMPLEMENTATION WITH VIVADO HLS

FELICS (Fair Evaluation of Lightweight Cryptographic System) project [19] provided source code. After simplifying source code, a top function which is named Encrypt performs the key schedule, pre-whitening, then in EncryptionPrinceCore function which performs ciphering. After the last round is finished, post-whitening is performed and ciphertext is obtained. To simulate the design, the main function which calls the top function was written. Later than, obtaining correct simulation, synthesis and implementation was performed.

The obtained design using Vivado HLS consumes 2278 LUTs, 484 FFs. Encryption takes up to 675 cycles and achieves a clock frequency of up to 219 MHz.

5.3. IMPLEMENTATION WITH PICOBLAZE

To compare results in terms of performance and area consumption, implementing PRINCE algorithm with a standard processor gives substantial information.

Required tools and files were downloaded from Xilinx website. To implement the PRINCE algorithm, the C code of the algorithm was converted into assembly code with the PicoBlaze instruction set. The PicoBlaze assembler created program memory for the processor with this assembly code. First, assembly code has started to be written with loop unrolled, but PRINCE algorithm has a lot of loops and functions. That is why to reduce consumption of resources, loops and functions were transferred to the code as they are. After all, program memory was generated using 735 instructions then synthesized and implemented with the source code of PicoBlaze.

With a clock frequency of up to 235 MHz, the design used 127 LUTs and 82 FFs, with a latency of 27629 clock cycles for an encryption operation.

5.4. IMPLEMENTATION WITH VSCPU

First of all, in this implementation pipelined VSCPU was used. Unlike the original design, flow mentioned previously, in this study VSCPU assembly code was written by hand. Because, C compiler is under development, the generated VSCPU assembly code is longer

than manually written code (*The codes can be found in Appendix A*). The rest of the flow is the same, handwritten VSCPU assembly code was simulated then generated machine code was inserted to RTL memory files. After that, all files are combined with VSCPU. Finally, the design was synthesized and verified with Xilinx ISE software.

Handwritten VSCPU assembly code includes 735 32-bit instructions. There are 3 memory elements; one of them is ROM for fetch instruction and the others are RAM to read and write data. The encryption takes 15483 clock cycles. Using 173 FFs and 922 LUTs, the design is capable of operating at a clock frequency of up to 131 MHz.

5.5. IMPLEMENTATION WITH 32-BIT CUSTOMIZED VSCPU

Handwritten VSCPU assembly code is shorter than C compiler-generated code, but PRINCE algorithm has lots of basic operations such as XOR, fixed shift, etc. For example, to execute an XOR operation, with the original VSCPU instruction set, NAND instruction is executed 4 times. This causes using more memory and taking up more cycles because every NAND instruction has to travel 4 states which are mentioned before. As previously stated VSCPU can be customized so adjusting the instruction set of VSCPU can be helpful to get better results.

Because of optimization concerns stated in the previous section, the instruction set was changed and the PRINCE algorithm was written manually with a new instruction set of VSCPU. First of all, in the PRINCE algorithm, the most executed operation is XOR. For this reason, XOR instruction was added. Due to the PRINCE algorithm structure, it has a lot of loops, and functions executed in loops can be rewritten into the VSCPU RTL design. So, the key expansion, S-layer, and linear layer functions were inserted to VSCPU design as RKPRIME, SR, INVSR, M01, and S01 (*The codes can be found in Appendix B*). Unused instructions removed. Tables 5.4 and 5.5 show the description of the new instruction set and assembly code.

Table 5.1. The new instruction set of 32-Bit Customized VSCPU

Mnemonic	**Description**	**Microoperation**
ADDi A B	unsigned add immediate	mem[A] ← mem[A] + B

Table 5.1. continued

CP A B	copy data	mem[A] ← mem[B]
CPI A B	copy data indirect	mem[A] ← mem[mem[B]]
BZJ A B	branch on zero	PC ← (mem[B] == 0) ? mem[A] : (PC + 1)
LTi A B	compare and set immediate	mem[A] ← (mem[A] < B) ? 1 : 0
XOR A B	bitwise xor	mem[A] ← mem[A] ^ mem[B]
SR A B	shift function	mem[A] ← {((mem[B][7:0] & 0xF0) ^ (mem[B][23:16] & 0x0F)),...,((mem[A][23:16] & 0xF0) ^ (mem[B][7:0] & 0x0F))} mem[B] ← {((mem[B][31:24] & 0xF0) ^ (mem[A][15:8] & 0x0F)),...,((mem[A][15:8] & 0xF0) ^ (mem[A][31:24] & 0x0F))}
M01 A B	m multiplication	mem[A] ← { (0x07 & (mem[B][7:0] >> 4)) ^ (0x0B & mem[B][7:0]) ^ (0x0D & (mem[B][15:8] >> 4)) ^ (0x0E & mem[B][15:8]),...} mem[B] ← (mem[A][31:24] << 4) ^ (0x0E & (mem[A][23:16] >> 4)) ^ (0x07 & mem[A][23:16]) ^ (0x0B & (mem[A][31:24] >> 4)) ^ (8'h0D & mem[A][31:24]),...}
S01 A B	sBox 0 or 1	mem[A] ← (B == 0) ? {s0[mem[A][31:24]],...} : {s1[mem[A][31:24]],...}

Table 5.1. continued

RKPRIME A B	create k0’	mem[B+1] ← {(((mem[A][23:16] << 7) & 0x80) ^ (mem[A][31:24] >> 1)),..,(((mem[B][31:24] << 7) & 0x80) ^ (mem[A][7:0] >> 1))} mem[B+2] ← {(((mem[B][23:16] << 7) & 0x80) ^ (mem[A][23:16] >> 1)),...(((mem[A][31:24] << 7) & 0x80) ^ (mem[B][7:0] >> 1))}
INVSR A B	inverse shift function	mem[A] ← {((mem[A][23:16] & 0xF0) ^ (mem[A][7:0] & 0x0F)),...,((mem[B][7:0] & 0xF0) ^ (mem[A][23:16] & 0x0F))} mem[B] ← {((mem[A][15:8] & 0xF0) ^ (mem[B][31:24] & 0x0F)),...,((mem[B][31:24] & 0xF0) ^ (mem[B][15:8] & 0x0F))}

Table 5.2. Assembly Code is written manually Customized 32-Bit VSCPU

1: RKPRIME 80 81	27: CPI 88 79	54: XOR 86 88	83: 0
//Whitening	28: XOR 87 88	55: ADDi 79 1	//K1
2: XOR 86 80	29: ADDi 79 1	56: CPI 88 79	84: 2003195204
3: XOR 87 81	30: XOR 86 84	57: XOR 87 88	85: 857870592
//ROUND Call	31: XOR 87 85	58: ADDi 79 1	//Block
4: CPI 88 79	//Whitening	59: XOR 86 84	86: 4023233417
5: XOR 86 88	32: XOR 86 82	60: XOR 87 85	87: 1732584193
6: ADDi 79 1	33: XOR 87 83	61: INVSR 86 87	88: 0
7: CPI 88 79	//ROUND	62: INVSR 86 87	//RC
8: XOR 87 88	34: S01 86 0	63: M01 86 87	91: 1148416003
9: ADDi 79 1	35: S01 87 0	64: M01 86 87	92: 780802323
10: XOR 86 84	36: M01 86 87	65: S01 86 1	93: 3492912937
11: XOR 87 85	37: M01 86 87	66: S01 87 1	94: 574097828
12: ADDi 73 14	38: SR 86 87	67: ADDi 77 1	95: 2305576684
13: BZJ 72 76	39: SR 86 87	68: CP 78 77	96: 2566532616
	40: CPI 88 79	69: LTi 78 5	97: 1997787192

Table 5.2. continued

14: CP 77 76	41: XOR 86 88	70: BZJ 75 78	98: 3860932677
//Mid Layer	42: ADDi 79 1	71: BZJ 74 76	99: 1812785460
15: S01 86 0	43: CPI 88 79	72: 34	100: 3479590078
16: S01 87 0	44: XOR 87 88	73: 0	101: 2975634941
17: M01 86 87	45: ADDi 79 1	74: 53	102: 2018506878
18: M01 86 87	46: XOR 86 84	75: 0	103: 2856561905
20: S01 87 1	47: XOR 87 85	76: 0	104: 1359512709
//INVERSE	48: ADDi 77 1	77: 0	105: 1413231141
ROUND CALL	49: CP 78 77	78: 0	106: 802390728
21: ADDi 75 23	50: LTi 78 5	79: 89	107: 224519136
22: BZJ 74 76	51: BZJ 73 78	//K0	108: 2500961636
23: CP 77 76	52: BZJ 72 76	80: 4293844428	109: 2569211082
24: CPI 88 79	//INVERSE	81: 3148519816	110: 2577642963
25: XOR 86 88	ROUND	//K0'	111: 3713039561
26: ADDi 79 1	53: CPI 88 79	82: 0	112: 3072961728

To generate machine code with assembly code, a new memory generator was written with Python language. After that, the assembly code size became much shorter than the original one.

This design of VSCPU assembly code includes 112 32-bit instructions. There are 3 memory elements; one of them is ROM for fetch instruction and the others are RAM to read and write data. The encryption takes 270 clock cycles. Using 215 FFs and 601 LUTs, the design is capable of operating at a clock frequency of up to 180 MHz.

5.6. IMPLEMENTATION WITH 64-BIT CUSTOMIZED VSCPU

Changing the instruction set of VSCPU gives better results. Because block and round keys sizes are 64 bits wide, there are some points that can be more optimized. In the previous design, some instructions had to be called back to back. For example, to create the k0’ key, RKPRIME instruction was called twice. Actually, at the first call VSCPU calculates k0’ prime key, but because of the VSCPU is 32 bits wide, the calculated second part of k0’ could not be transferred into memory. This architectural constraint causes assembly code to

become larger and spend more time on calculation. When the design is converted from 32 bits to 64 bits, INVSR instruction can be embedded into SR instruction. As shown in Table 5.2., the assembly code amount is significantly reduced.

At this point, we have three different implementations for block size bit wide VSCPU. Two of them are non-pipelined and the third one is pipelined. Non-pipelined implementations vary with their architecture Harvard and Von Neumann. All implementations are used with the same ISA.

First of all, assembly code generated manually for non-pipelined which was designed by Von Neumann Architecture. Later, by nature of Harvard architecture and pipelined design assembly code is separated and optimized.

Table 5.3. The instruction set of 64-Bit Customized VSCPU

Mnemonic	**Description**	**Microoperation**
ADDi A B	unsigned add	mem[A] ← mem[A] + B
LTi A B	compare and set immediate	mem[A] ← (mem[A] < B) ? 1 : 0
XOR A B	bitwise xor	mem[A] ← mem[A] ^ mem[B]
CP A B	copy data	mem[A] ← mem[B]
CPI A B	copy data indirect	mem[A] ← mem[mem[B]]
BZJ A B	branch on zero	PC ← (mem[B] == 0) ? mem[A] : (PC + 1)
SR A B	shift and inverse shift function	mem[A] ← (mem[B] == 0) ? {((mem[A][31:24] & 8'hF0) ^ (mem[A][47:40] & 8'h0F)),...,((mem[A][7:0] & 8'hF0) ^ (mem[A][23:16] & 8'h0F)) } : {(mem[A][31:24] & 8'hF0) ^ (mem[A][15:8] & 8'h0F),...,(mem[A][7:0] & 8'hF0) ^ (mem[A][55:48] & 8'h0F)}

Table 5.3. continued

M01 A B	m multiplication	mem[A] ← { (0x07 & (mem[B][7:0] >> 4)) ^ (0x0B & mem[B][7:0]) ^ (0x0D & (mem[B][15:8] >> 4)) ^ (0x0E & mem[B][15:8]),...,(mem[A][31:24] << 4) ^ (0x0E & (mem[A][23:16] >> 4)) ^ (0x07 & mem[A][23:16]) ^ (0x0B & (mem[A][31:24] >> 4)) ^ (8'h0D & mem[A][31:24]),...}
S01 A B	sBox	mem[A] ← (mem[B] == 0) ? {((s0[(63 - ((mem[A][63:56] >> 4) * 4)) -: 4] << 4) ^ (s0[63 - ((mem[A][63:56] & 8'h0F) * 4) -: 4])),...,((s0[(63 - ((mem[A][7:0] >> 4) * 4)) -: 4] << 4) ^ (s0[63 - ((mem[A][7:0] & 8'h0F) * 4) -: 4]))} : {((s1[(63 - ((mem[A][63:56] >> 4) * 4)) -: 4] << 4) ^ (s1[63 - ((mem[A][63:56] & 8'h0F) * 4) -: 4])),...,((s1[(63 - ((mem[A][7:0] >> 4) * 4)) -: 4] << 4) ^ (s1[63 - ((mem[A][7:0] & 8'h0F) * 4) -: 4]))}
RKPRIME A B	create k0’	mem[A] ← {(((mem[A][23:16] << 7) & 0x80) ^ (mem[A][31:24] >> 1)),..,(((mem[B][31:24] << 7) & 0x80) ^ (mem[A][7:0] >> 1))} mem[B+2] ← {(((mem[B][23:16] << 7) & 0x80) ^ (mem[A][23:16] >> 1)),...(((mem[A][31:24] << 7) & 0x80) ^ (mem[B][7:0] >> 1))}

Table 5.4. Assembly code is written manually Customized 64-Bit VSCPU

0: RKPRIME 56 57	21: LTi 41 5	42: 0

Table 5.4. continued

1: XOR 55 56	22: BZJ 38 41	43: 0
2: XOR 55 43	23: CPI 42 43	44: 4932409175868840211
3: ADDi 43 44	24: XOR 55 42	45: 15001946832764406180
4: XOR 55 58	25: XOR 55 58	46: 9902376458766659080
5: CP 41 39	26: ADDi 43 1	47: 8580430657868605509
6: LTi 41 5	27: SR 55 1	48: 7785854268843906238
7: BZJ 37 41	28: M01 55 1	49: 12780254758448396414
8: S01 55 0	29: S01 55 1	50: 12268839962333971589
9: M01 55 1	30: ADDi 40 1	51: 6069781533086155464
10: SR 55 0	31: BZJ 60 36	52: 964302348947137892
11: CPI 42 43	32: CPI 42 43	53: 11034677576288417235
12: XOR 55 42	33: XOR 55 42	54: 15947383486322158784
13: XOR 55 58	34: XOR 55 58	55: 17279655951921914625
14: ADDi 43 1	35: XOR 55 57	56: 18441921395520346504
15: ADDi 39 1	36: 0	57: 0
16: BZJ 59 36	37: 17	58: 8603657889541918976
17: S01 55 0	38: 32	59: 5
18: M01 55 1	39: 0	60: 20
19: S01 55 1	40: 0	
20: CP 41 40	41: 0	

Table 5.4. shows back-to-back instruction calls that are not necessary anymore.

The design of the non-pipelined Von Neumann VSCPU assembly code includes 61 64-bit instructions and data. There is one memory element to fetch, read and write data. The encryption takes 562 clock cycles. Using 114 FFs and 662 LUTs, the design is capable of operating at a clock frequency of up to 259 MHz.

The design of the non-pipelined Harvard VSCPU assembly code includes 36 14-bit instructions and 25 64-bit data. There are two memory elements, one of them is ROM for fetch instruction and the other one is RAM to read and write data. The encryption takes 562

clock cycles. Using 102 FFs and 663 LUTs, the design is capable of operating at a clock frequency of up to 259 MHz.

The design of pipelined VSCPU assembly code includes 36 14-bit instructions and 25 64-bit data. There are 3 memory elements; one of them is ROM for fetch instruction and the others are RAM to read and write data. The encryption takes 186 clock cycles. Using 236 FFs and 723 LUTs, the design is capable of operating at a clock frequency of up to 249 MHz.

5.7. EVALUATION OF PRINCE IMPLEMENTATIONS

The main purpose of this study is to achieve optimization of latency and resource usage in the Prince block cipher algorithm implementation. To achieve that aim, the algorithm implemented several ways

Table 5.5. Memory related information

Implementations	**Instruction Bit Length**	**Number of Instruction**	**Code (Byte)**	**Data Bit Length**	**Number of Data**	**Data (Byte)**
PicoBlaze	18	770	1733	8	-	-
VSCPU	32	658 * 3	7896	32	77 * 3	924
VSCPU-32-P-C	32	72 * 3	864	32	41 * 3	492
VSCPU-64-NP-V	64	36	288	64	25	200
VSCPU-64-NP-H	14	36	63	64	25	200
VSCPU-64-P	14	36	63	64	25 * 2	400
AVR	-	-	1394	-	-	78
MSP	-	-	1576	-	-	76
ARM	-	-	1364	-	-	284

Table 5.6. Resource related information

Implementations	LUT	FF	Max Clock Frequency (MHz)	Latency (cyc)
Pure-RTL	1880	241	68.731	20
Vivado HLS	2278	484	219.490	675
PicoBlaze	127	82	235.449	27629
VSCPU	922	173	131.119	15482
VSCPU-32-P-C	601	215	180.573	270
VSCPU-64-NP-V	662	114	259.410	562
VSCPU-64-NP-H	663	102	259.410	562
VSCPU-64-P	723	236	249.433	186

Table 5.7. Cycle related information

Implementations	Latency (Cycle)
Pure-RTL	20
Vivado HLS	675
PicoBlaze	27629
VSCPU	15482
VSCPU-32-P-C	270
VSCPU-64-NP-V	562
VSCPU-64-NP-H	562
VSCPU-64-P	186
AVR [13]	19930
MSP [13]	28306
ARM [13]	14817

Table 5.8. Time related information

Implementations	Latency (ns)
Pure-RTL	290.94
Vivado HLS	3075.30
PicoBlaze	117340.36
VSCPU	118081.21
VSCPU-32-P-C	1495.26
VSCPU-64-NP-V	2166.51
VSCPU-64-NP-H	2166.51
VSCPU-64-P	746.418

Pure-RTL design achieves encryption within 1 cycle with 20 cycle latency. This is the best latency result. In contrast to the timing success of Pure-RTL, it was designed and validated manually. That is why it takes more time than other works to design. Due to the solid structure of its design, it is impossible to develop software over Pure-RTL.

Implementing with Vivado HLS cuts down on design time tremendously. But, timing and resource usage results show that implementing with Vivado HLS is not as good as Pure-RTL. Completing encryption operation takes 675 cycles, which is almost 33 times more than the number of cycles in Pure-RTL. Also, LUT and FF consumption increased almost twice. Furthermore, implementation with Vivado HLS does not allow programming over the design, because it does not include a processor just like Pure-RTL design.

Developing assembly code to run over PicoBlaze's processor is another way to implement the Prince algorithm. Contrary to expectations, consumption of resources was significantly reduced. However, the obtained results show that using PicoBlaze takes a huge amount of time. PicoBlaze implementation completes the encryption operation within 27629 cycles and it is 1381 times more than Pure-RTL.

In terms of resource consumption and timing balance, the original VSCPU is the worst implementation. VSCPU has a very limited instruction set that is not compatible with the

Prince algorithm. That is why code size increases and it affects timing and resource consumption. It uses less LUT and FF but the operation is completed within 15482 cycles.

The first customization of VSCPU was changing its instruction set. Implementation algorithm with 32-Bit customized VSCPU gives promising results. Thanks to customization of VSCPU, it is completed within 270 cycles and total FF and LUT consumption reduced.

As table 5.6. shows above, there are 3 different types of implementation for 64-bit Customized VSCPU. Two of them are non-pipelined and one of them is pipelined. The results of the non-pipelined implementations are promising in terms of resource consumption. The amount of LUT and FF consumptions of non-pipelined Harvard 64-Bit, non-pipelined Von Neumann 64-Bit, and pipelined 64-Bit are 765, 776, and 959 respectively. Although non-pipelined versions of VSCPU are better, pipelined VSCPU is almost 3 times faster.

When compare 64-Bit pipelined VSCPU result with the implementation of other works, in 2019, Daniel Dinu et al. [13] implemented the Prince algorithm with different scenarios and platforms. Implementing with 64-Bit pipelined VSCPU has less code size. In addition, implementation with MSP, ARM, and AVR takes more time to complete encryption operation.

As a result of this implementation, the pipelined 64-Bit VSCPU gives the best result in terms of the balance of resource consumption and timing.

6. CONCLUSION

The number of IoT devices around the people is increasing day by day. Information security has become a significant problem with these devices that enter all areas of the lives of humans to provide convenience. Thanks to the Lightweight cryptographic algorithms developed in line with the limits of the integrated circuit of IoT devices, beneficial security is provided within these limits.

The aim of this study is to try to provide security in the most optimized way in IoT applications that are challenging due to the mentioned limits. Accordingly, the Prince lightweight block cipher algorithm was implemented using Pure-RTL, Vivado HLS, PicoBlaze, VSCPU and modified VSCPU. The obtained results were compared among themselves and with the results of previous studies.

As mentioned in the previous section, 64-Bit pipelined VSCPU, which was developed by making necessary changes in the instruction set and bit width of the original VSCPU, is the optimum solution in terms of resource usage and timing.

REFERENCES

1. Simmons G. Cryptology. Encyclopedia Britannica. [cited 10 June 2021]. Available from: https://www.britannica.com/topic/cryptology

2. Paar, Christof, and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media; 2009.

3. What is the Internet of Things (IoT)? Oracle.com [cited 13 June 2021]. Available from: https://www.oracle.com/tr/internet-of-things/what-is-iot/

4. Daemen J. Rijmen V. AES proposal: Rijndael. 1999.

5. National Bureau of Standards, US Department of Commerce. Federal information processing standards publication: data encryption standard (DES). 1993.

6. Thakor V, Razzaque M, Khandaker M. Lightweight Cryptography algorithms for resource-constrained IoT devices: a review, comparison and research opportunities. *IEEE Access*. 2021;9:28177-28193.

7. What is an FPGA? Field Programmable Gate Array. Xilinx. 2021 [cited 22 June 2021]. Available from: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

8. Eastland N. Structure of an FPGA. Digilent Blog. 2021 [cited 22 June 2021]. Available from: https://blog.digilentinc.com/structure-of-an-fpga

9. Bilgili B., Yamaneren C, Vatansever K, Çoltu U., Vivado yüksek seviyeli tasarım aracı kullanılarak kırmık üstü sistem tasarımı [Thesis]. Istanbul: Istanbul Technical University; 2019.

10. UG129, X. U. *PicoBlaze 8-bit Embedded Microcontroller User Guide. V2. 1*. Xilinx; 2011.

11. Yildiz A, Ugurdag H, Aktemur B, Iskender D, Goren S. CPU design simplified. *2018 3rd International Conference on Computer Science and Engineering (UBMK)*. 2018:630-632. IEEE.

12. Varici A, Saglam G, Ipek S, Yildiz A, Goren S, Aysu A et al. Fast and efficient implementation of lightweight crypto algorithm Present on FPGA through processor instruction set extension. *2019 IEEE East-West Design & Test Symposium (EWDTS)*. 2019.

13. Dinu D, Corre Y, Khovratovich D, Perrin L, Großschädl J, Biryukov A. Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering*. 2018;9(3):283-302.

14. Diehl W, Farahmand F, Yalla P, Kaps J, Gaj K. Comparison of hardware and software implementations of selected lightweight block ciphers. *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017.

15. Abbas Y, Jidin R, Jamil N, Z'aba M, Rusli M, Tariq B. Implementation of Prince algorithm in FPGA. Proceedings of the *6th International Conference on Information Technology and Multimedia*. 2014.

16. What are Substitution-Permutation networks? Educative: Interactive Courses for Software Developers. 2021 [cited 2 July 2021]. Available from: https://www.educative.io/edpresso/what-are-substitution-permutation-networks

17. Borghoff J, Canteaut A, Güneysu T, Kavun E, Knezevic M, Knudsen L et al. Prince – a low-latency block cipher for pervasive computing applications. *ASIACRYPT*. 2012:208-255.

18. 9. GitHub - huljar/prince-vhdl: Implementation of the Prince lightweight block cipher in VHDL. GitHub. 2021 [cited 10 July 2021]. Available from: https://github.com/huljar/prince-vhdl

19. Dinu, D., Biryukov, A., Großschädl, J., Khovratovich, D., Le Corre, Y., & Perrin, L. Felics–fair evaluation of lightweight cryptographic systems. *In NIST Workshop on Lightweight Cryptography* volume 128. 2015.

APPENDIX A: HANDWRITTEN 32-BIT VSCPU ASSEMBLY CODE

```
0: CP 224 220
1: CP 225 221
2: NAND 224 170
3: NAND 224 224
4: NAND 225 170
5: NAND 225 225
6: CP 226 220
7: CP 227 221
8: SRLi 226 40
9: SRLi 227 40
10: SRLi 224 24
11: SRLi 225 24
12: NAND 226 226
13: NAND 225 225
14: NAND 226 225
15: NAND 227 227
16: NAND 224 224
17: NAND 227 224
18: SRLi 226 39
19: SRLi 227 39
20: NAND 226 171
21: NAND 226 226
22: NAND 227 171
23: NAND 227 227
170: 4278190080
171: 2155905152
172: 2139062143
173: 16777215
218: 4023233417
219: 1732584193
220: 4293844428
221: 3148519816
222: 2003195204
223: 857870592
290: 291
291: CPI 148 146
292: ADDi 146 1
293: CPI 149 146
294: CP 150 218
295: NAND 150 148
296: NAND 218 150
297: NAND 150 148
298: NAND 218 150
299: CP 150 219
300: NAND 150 149
301: NAND 219 150
302: NAND 150 149
303: NAND 219 150
304: ADDi 146 1
305: CP 150 218
427: NAND 156 151
428: CP 151 156
429: NAND 151 155
430: NAND 156 151
431: NAND 151 155
432: NAND 156 151
433: CP 151 150
434: CP 152 150
435: CP 154 153
436: CP 155 153
437: SRLi 156 36
438: SRLi 151 4
439: NANDi 151 11
440: NAND 151 151
441: NANDi 152 13
442: NAND 152 152
443: SRLi 154 4
444: NANDi 154 14
445: NAND 154 154
446: NANDi 155 7
447: NAND 155 155
448: CP 157 156
565: NAND 156 151
566: CP 151 150
567: CP 152 150
568: CP 154 153
569: CP 155 153
570: SRLi 156 36
571: SRLi 151 4
572: NANDi 151 13
573: NAND 151 151
574: NANDi 152 14
575: NAND 152 152
576: SRLi 154 4
577: NANDi 154 7
578: NAND 154 154
579: NANDi 155 11
580: NAND 155 155
581: CP 157 156
582: NAND 157 151
583: NAND 156 157
584: NAND 157 151
585: NAND 156 157
586: CP 157 156
718: CP 154 160
719: CP 155 162
720: CP 160 164
721: CPi 148 160
722: CPi 149 162
723: CPi 153 725
724: BZJi 122 688
725: CP 162 166
726: CPi 148 162
727: CPi 149 164
728: CPi 153 730
729: BZJi 122 688
730: CP 164 154
731: CPi 148 164
732: CPi 149 166
733: CPi 153 735
734: BZJi 122 688
735: CP 166 155
736: CPi 148 166
737: CPi 149 154
738: CPi 153 740
739: BZJi 122 688
740: BZJ 147 122
741: SRLi 160 56
742: SRLi 161 48
743: SRLi 162 40
744: SRLi 164 56
745: SRLi 165 48
746: SRLi 166 40
```

24: NAND 226 171	306: NAND 150 222	449: NAND 157 151	587: NAND 157 152	747: NAND 160 160
25: NAND 226 226	307: NAND 218 150	450: NAND 156 157	588: NAND 156 157	748: NAND 161 161
26: NAND 227 171	308: NAND 150 222	451: NAND 157 151	589: NAND 157 152	749: NAND 160 161
27: NAND 227 227	309: NAND 218 150	452: NAND 156 157	590: NAND 156 157	750: NAND 160 160
28: CP 224 220	310: CP 150 219	453: CP 157 156	591: CP 157 156	751: NAND 162 162
29: CP 225 221	311: NAND 150 223	454: NAND 157 152	592: NAND 157 154	752: NAND 160 162
30: SRLi 224 1	312: NAND 219 150	455: NAND 156 157	593: NAND 156 157	753: NAND 160 160
31: SRLi 225 1	313: NAND 150 223	456: NAND 157 152	594: NAND 157 154	754: NAND 163 163
32: NAND 224 172	314: NAND 219 150	457: NAND 156 157	595: NAND 156 157	755: NAND 160 163
33: NAND 224 224	315: BZJi 147 0	458: CP 157 156	596: CP 157 156	756: CP 218 160
34: NAND 225 172	316: CPi 162 0	459: NAND 157 154	597: NAND 157 155	757: NAND 164 164
35: NAND 225 225	317: CPi 149 8	460: NAND 156 157	598: NAND 156 157	758: NAND 165 165
36: CP 228 226	318: NAND 149 149	461: NAND 157 154	599: NAND 157 155	759: NAND 164 165
37: CP 229 227	319: ADDi 149 1	462: NAND 156 157	600: NAND 156 157	760: NAND 164 164
38: NAND 226 224	320: CPi 148 218	463: CP 157 156	601: CP 151 150	761: NAND 166 166
39: NAND 227 225	321: CPi 150 0	464: NAND 157 155	602: CP 152 150	762: NAND 164 166
40: NAND 228 226	322: CPi 151 0	465: NAND 156 157	603: CP 154 153	763: NAND 164 164
41: NAND 229 227	323: CPi 154 362	466: NAND 157 155	604: CP 155 153	764: NAND 167 167
42: NAND 226 224	324: CPi 155 331	467: NAND 156 157	605: SRLi 151 4	765: NAND 164 167
43: NAND 227 225	325: CPi 163 326	468: CP 151 150	606: NANDi 151 14	766: CP 219 164
44: NAND 228 226	326: CP 152 150	469: CP 152 150	607: NAND 151 151	767: BZJ 147 122
45: NAND 229 227	327: CP 156 170	470: CP 154 153	608: NANDi 152 7	
46: CP 230 228	328: CPi 158 24	471: CP 155 153	609: NAND 152 152	
	329: LTi 152 2		610: SRLi 154 4	
	330: BZJ 147 152			
	331: CP 153 151			
	332: LTi 153 4			
	333: BZJ 154 153			
	334: CPI 157 148			

47: NAND 230 170	335: NAND 157 156	472: SRLi 151 4	611: NANDi 154 11	768: CP 154 161
48: NAND 230 230	336: NAND 157 157	473: NANDi 151 13	612: NAND 154 154	769: CPi 148 161
49: CP 231 221	337: SRLi 156 8	474: NAND 151 151	613: NANDi 155 13	770: CPi 149 163
50: NANDi 231 255	338: SRL 157 158	475: NANDi 152 14	614: NAND 155 155	771: CPi 153 773
51: NAND 231 231	339: CP 159 157	476: NAND 152 152	615: CP 157 151	772: BZJi 122 688
52: SRLi 231 7	340: SRLi 157 4	477: SRLi 154 4	616: NAND 151 152	773: CPi 148 163
53: NANDi 231 1	341: ADD 157 169	478: NANDi 154 7	617: NAND 157 151	774: CPi 149 165
54: NAND 231 231	342: CPI 157 157	479: NAND 154 154	618: NAND 151 152	775: CPi 153 777
55: SRLi 231 56	343: SRLi 157 36	480: NANDi 155 11	619: NAND 157 151	776: BZJi 122 688
56: CP 232 230	344: NANDi 159 15	481: NAND 155 155	620: CP 151 157	777: CPi 148 165
57: NAND 230 231	345: NAND 159 159	482: CP 157 151	621: NAND 151 154	778: CPi 149 167
58: NAND 232 230	346: ADD 159 169	483: NAND 151 152	622: NAND 157 151	779: CPi 153 781
59: NAND 230 231	347: CPI 159 159	484: NAND 157 151	623: NAND 151 154	780: BZJi 122 688
60: NAND 232 230	348: CP 160 157	485: NAND 151 152	624: NAND 157 151	781: CPi 148 167
61: NAND 228 173	349: NAND 160 159	486: NAND 157 151	625: CP 151 157	782: CPi 149 154
62: NAND 228 228	350: NAND 157 160	487: CP 151 157	626: NAND 151 155	783: CPi 153 785
63: NAND 232 232	351: NAND 160 159	488: NAND 151 154	627: NAND 157 151	784: BZJi 122 688
64: NAND 228 228	352: NAND 157 160	489: NAND 157 151	628: NAND 151 155	785: CP 154 166
65: NAND 228 232	353: CP 161 158	490: NAND 151 154	629: NAND 157 151	786: CP 155 164
66: CP 224 228	354: ADDi 161 32	491: NAND 157 151	630: CP 151 150	787: CP 166 162
67: CP 225 229	355: SRL 157 161	492: CP 151 157	631: CP 152 150	788: CPi 148 166
68: CP 226 218	356: NAND 162 162	493: NAND 151 155	632: CP 154 153	789: CPi 149 164
69: NAND 226 220	357: NAND 157 157		633: CP 155 153	790: CPi 153 792
	358: NAND 162 157			791: BZJi 122 688
	359: ADD 158 149			792: CP 164 160
	360: ADDi 151 1			793: CPi 148 164
	361: BZJ 155 122			794: CPi 149 162
	362: CPIi 148 162			795: CPi 153 797
				796: BZJi 122 688
				797: CP 162 154
				798: CPi 148 162
				799: CPi 149 160
				800: CPi 153 802
				801: BZJi 122 688
				802: CP 160 155
				803: CPi 148 160
				804: CPi 149 154
				805: CPi 153 807
				806: BZJi 122 688

70: NAND 218 226
71: NAND 226 220
72: NAND 218 226
73: CP 226 219
74: NAND 226 221
75: NAND 219 226
76: NAND 226 221
77: NAND 219 226
78: CPi 856 80
79: BZJi 122 857
80: CP 226 218
81: NAND 226 224
82: NAND 218 226
83: NAND 226 224
84: NAND 218 226
85: CP 226 219
86: NAND 226 225
87: NAND 219 226
88: NAND 226 225
89: NAND 219 226
90: 11
91: 15
92: 3
93: 2
363: CPi 162 0
364: CPi 151 0
365: CPi 148 219
366: ADDi 150 1
367: BZJ 163 122
368: 369
369: CPi 151 8
370: NAND 151 151
371: ADDi 151 1
372: CPi 153 160
373: CPi 154 0
374: CPi 155 0
375: CPi 158 385
376: CPi 159 379
377: CPi 168 398
378: CPi 149 218
379: CP 156 154
380: CP 150 170
381: LTi 156 2
382: BZJ 687 156
383: CPi 152 24
384: CPi 155 0
385: CP 157 155
386: LTi 157 4
387: CPI 148 149
388: BZJ 168 157
389: NAND 148 150
390: NAND 148 148
391: SRL 148 152
392: SRLi 150 8
393: ADD 152 151
394: CPIi 153 148
395: ADDi 153 1
396: ADDi 155 1
397: BZJ 158 122
398: ADDi 154 1
494: NAND 157 151
495: NAND 151 155
496: NAND 157 151
497: CP 151 150
498: CP 152 150
499: CP 154 153
500: CP 155 153
501: SRLi 157 36
502: SRLi 151 4
503: NANDi 151 14
504: NAND 151 151
505: NANDi 152 7
506: NAND 152 152
507: SRLi 154 4
508: NANDi 154 11
509: NAND 154 154
510: NANDi 155 13
511: NAND 155 155
512: CP 158 157
513: NAND 158 151
514: NAND 157 158
515: NAND 158 151
516: NAND 157 158
517: CP 158 157
634: SRLi 157 36
635: SRLi 151 4
636: NANDi 151 7
637: NAND 151 151
638: NANDi 152 11
639: NAND 152 152
640: SRLi 154 4
641: NANDi 154 13
642: NAND 154 154
643: NANDi 155 14
644: NAND 155 155
645: CP 158 157
646: NAND 158 151
647: NAND 157 158
648: NAND 158 151
649: NAND 157 158
650: CP 158 157
651: NAND 158 152
652: NAND 157 158
653: NAND 158 152
654: NAND 157 158
655: CP 158 157
807: BZJ 147 122
808: 0
809: 0
810: 0
811: CPi 169 90
812: CP 810 809
813: LTi 810 5
814: BZJ 808 810
815: CPi 147 817
816: BZJi 122 316
817: CPi 147 819
818: BZJi 122 668
819: CPi 147 821
820: BZJi 122 701
821: CPi 147 823
822: BZJi 122 741
823: CPi 147 825
824: BZJi 122 291
825: ADDi 809 1
826: BZJi 122 812
827: CPi 147 829
828: BZJi 122 316
829: CPi 147 831
830: BZJi 122 668
831: CPi 147 833
832: BZJi 122 741
833: CPi 169 106
834: CPi 147 836
835: BZJi 122 316
836: BZJ 808 122
837: 0
838: 0
839: CP 838 837
840: LTi 838 5

94: 10
95: 12
96: 9
97: 1
98: 6
99: 7
100: 8
101: 0
102: 14
103: 5
104: 13
105: 4
106: 11
107: 7
108: 3
109: 2
110: 15
111: 13
112: 8
113: 9
114: 10
115: 6
116: 4
117: 0
118: 5
119: 14
120: 12
121: 1
122: 0
123: 0
124: 1148416003
125: 780802323
126: 3492912937
127: 574097828
128: 2305576684
129: 2566532616
130: 1997787192
131: 3860932677
132: 1812785460

399: ADDi 149 1
400: BZJ 159 122
401: 402
402: CPI 150 149
403: CP 151 150
404: CP 152 150
405: CPI 153 148
406: CP 154 153
407: CP 155 153
408: SRLi 151 4
409: NANDi 151 7
410: NAND 151 151
411: NANDi 152 11
412: NAND 152 152
413: SRLi 154 4
414: NANDi 154 13
415: NAND 154 154
416: NANDi 155 14
417: NAND 155 155
418: CP 156 151
419: NAND 151 152
420: NAND 156 151
421: NAND 151 152
422: NAND 156 151
423: CP 151 156
424: NAND 151 154

518: NAND 158 152
519: NAND 157 158
520: NAND 158 152
521: NAND 157 158
522: CP 158 157
523: NAND 158 154
524: NAND 157 158
525: NAND 158 154
526: NAND 157 158
527: CP 158 157
528: NAND 158 155
529: NAND 157 158
530: NAND 158 155
531: NAND 157 158
532: CPIi 149 156
533: CPIi 148 157
534: BZJ 687 122
535: CPI 150 149
536: CP 151 150
537: CP 152 150
538: CPI 153 148
539: CP 154 153
540: CP 155 153
541: SRLi 151 4
542: NANDi 151 11

656: NAND 158 154
657: NAND 157 158
658: NAND 158 154
659: NAND 157 158
660: CP 158 157
661: NAND 158 155
662: NAND 157 158
663: NAND 158 155
664: NAND 157 158
665: CPIi 149 156
666: CPIi 148 157
667: BZJ 687 122
668: CPi 687 670
669: BZJi 122 369
670: CPi 148 166
671: CPi 149 167
672: CPi 687 674
673: BZJi 122 402
674: CPi 148 164
675: CPi 149 165
676: CPi 687 678
677: BZJi 122 535
678: CPi 148 162
679: CPi 149 163
680: CPi 687 682
681: BZJi 122 535
682: CPi 148 160
683: CPi 149 161
684: CPi 687 686
685: BZJi 122 402
686: BZJ 147 122

841: BZJ 808 838
842: CPi 147 844
843: BZJi 122 291
844: CPi 687 846
845: BZJi 122 369
846: CPi 147 848
847: BZJi 122 768
848: CPi 147 850
849: BZJi 122 670
850: CPi 147 852
851: BZJi 122 741
852: CPi 147 854
853: BZJi 122 316
854: ADDi 837 1
855: BZJi 122 839
856: 0
857: CPi 147 859
858: BZJ 290 122
859: CPi 808 861
860: BZJi 122 811
861: CPi 808 863
862: BZJi 122 827
863: CPi 808 865
864: BZJi 122 839
865: CPi 147 867
866: BZJ 290 122
867: BZJ 856 122

133: 3479590078 134: 2975634941 135: 2018506878 136: 2856561905 137: 1359512709 138: 1413231141 139: 802390728 140: 224519136 141: 2500961636 142: 2569211082 143: 2577642963 144: 3713039561 145: 3072961728 146: 122	425: NAND 156 151 426: NAND 151 154	543: NAND 151 151 544: NANDi 152 13 545: NAND 152 152 546: SRLi 154 4 547: NANDi 154 14 548: NAND 154 154 549: NANDi 155 7 550: NAND 155 155 551: CP 156 151 552: NAND 151 152 553: NAND 156 151 554: NAND 151 152 555: NAND 156 151 556: CP 151 156 557: NAND 151 154 558: NAND 156 151 559: NAND 151 154 560: NAND 156 151 561: CP 151 156 562: NAND 151 155 563: NAND 156 151 564: NAND 151 155	687: 0 688: CPI 150 148 689: NANDi 150 240 690: NAND 150 150 691: CPI 151 149 692: NANDi 151 15 693: NAND 151 151 694: CP 152 150 695: NAND 152 151 696: NAND 150 152 697: NAND 152 151 698: NAND 150 152 699: CPIi 148 150 700: BZJi 153 0 701: CP 154 167 702: CPi 148 167 703: CPi 149 165 704: CPi 153 706 705: BZJi 122 688 706: CPi 148 165 707: CPi 149 163 708: CPi 153 710 709: BZJi 122 688 710: CPi 148 163 711: CPi 149 161 712: CPi 153 714 713: BZJi 122 688 714: CPi 148 161 715: CPi 149 154 716: CPi 153 718 717: BZJi 122 688	

APPENDIX B: THE RTL FOR 32-BIT CUSTOMIZED VSCPU

```
`SR: begin
        if(SRCounterCurrent == 0)begin
                b0 = operand_a_actual_data[31:24]; //block[0]
                b1 = operand_a_actual_data[23:16]; //block[1]
                b2 = operand_a_actual_data[15:8] ; //block[2]
                b3 = operand_a_actual_data[7:0]  ; //block[3]
                b4 = operand_b_actual_data[31:24]; //block[4]
                b5 = operand_b_actual_data[23:16]; //block[5]
                b6 = operand_b_actual_data[15:8] ; //block[6]
                b7 = operand_b_actual_data[7:0]  ; //block[7]
                b7_ = b7;
                b7 = (b7 & 8'hF0) ^ (b5 & 8'h0F);
                b5 = (b5 & 8'hF0) ^ (b3 & 8'h0F);
                b3 = (b3 & 8'hF0) ^ (b1 & 8'h0F);
                b1 = (b1 & 8'hF0) ^ (b7_ & 8'h0F);
                b0_ = b0;
                b2_ = b2;
                b0 = (b4 & 8'hF0) ^ (b2 & 8'h0F);
                b2 = (b6 & 8'hF0) ^ (b4 & 8'h0F);
                b4 = (b0_ & 8'hF0) ^ (b6 & 8'h0F);
                b6 = (b2_ & 8'hF0) ^ (b0_ & 8'h0F);
                block0Next = {b0,b1,b2,b3};
```

```
                block1Next = {b4,b5,b6,b7};
                result_data = block0Next;
                result_addr = instruction_word_in[27:14];
                SRCounterNext = 1;
        end
        else if (SRCounterCurrent == 1) begin
                result_data = block1Current;
                result_addr = instruction_word_in[13:0];
                SRCounterNext = 0;
        end
        result_wr_en = 1;
end
`INVSR: begin
        if(SRCounterCurrent == 0)begin
                b0 = operand_a_actual_data[31:24];
                b1 = operand_a_actual_data[23:16];
                b2 = operand_a_actual_data[15:8] ;
                b3 = operand_a_actual_data[7:0]  ;
                b4 = operand_b_actual_data[31:24];
                b5 = operand_b_actual_data[23:16];
                b6 = operand_b_actual_data[15:8] ;
                b7 = operand_b_actual_data[7:0]  ;
                b1_ = b1;
                b1 = (b1 & 8'hF0) ^ (b3 & 8'h0F);
                b3 = (b3 & 8'hF0) ^ (b5 & 8'h0F);
```

```
                b5 = (b5 & 8'hF0) ^ (b7 & 8'h0F);
                b7 = (b7 & 8'hF0) ^ (b1_ & 8'h0F);
                b6_ = b6;
                b4_ = b4;
                b6 = (b2 & 8'hF0) ^ (b4 & 8'h0F);
                b4 = (b0 & 8'hF0) ^ (b2 & 8'h0F);
                b2 = (b6_ & 8'hF0) ^ (b0 & 8'h0F);
                b0 = (b4_ & 8'hF0) ^ (b6_ & 8'h0F);
                block0Next = {b0,b1,b2,b3};
                block1Next = {b4,b5,b6,b7};
                result_data = block0Next;
                result_addr = instruction_word_in[27:14];
                SRCounterNext = 1;
        end
        else if (SRCounterCurrent == 1) begin
                result_data = block1Current;
                result_addr = instruction_word_in[13:0];
                SRCounterNext = 0;
        end
        result_wr_en = 1;
end
`M01: begin
        if (SRCounterCurrent == 0) begin
                b0 = operand_a_actual_data[31:24];
                b1 = operand_a_actual_data[23:16];
```

```
b2 = operand_a_actual_data[15:8] ;

b3 = operand_a_actual_data[7:0]  ;

b4 = operand_b_actual_data[31:24];

b5 = operand_b_actual_data[23:16];

b6 = operand_b_actual_data[15:8] ;

b7 = operand_b_actual_data[7:0]  ;

b7_ = (8'h07 & (b7 >> 4)) ^ (8'h0B & b7) ^ (8'h0D & (b6 >> 4)) ^ (8'h0E & b6);

b7_ = (b7_ << 4) ^ (8'h0B & (b7 >> 4)) ^ (8'h0D & b7) ^ (8'h0E & (b6 >> 4)) ^ (8'h07 &
b6);

b6_ = (8'h0D & (b7 >> 4)) ^ (8'h0E & b7) ^ (8'h07 & (b6 >> 4)) ^ (8'h0B & b6);

b6_ = (b6_ << 4) ^ (8'h0E & (b7 >> 4)) ^ (8'h07 & b7) ^ (8'h0B & (b6 >> 4)) ^ (8'h0D &
b6);

b7 = b7_;

b6 = b6_;

b5_ = (8'h0B & (b5 >> 4)) ^ (8'h0D & b5) ^ (8'h0E & (b4 >> 4)) ^ (8'h07 & b4);

b5_ = (b5_ << 4) ^ (8'h0D & (b5 >> 4)) ^ (8'h0E & b5) ^ (8'h07 & (b4 >> 4)) ^ (8'h0B &
b4);

b4_ = (8'h0E & (b5 >> 4)) ^ (8'h07 & b5) ^ (8'h0B & (b4 >> 4)) ^ (8'h0D & b4);

b4_ = (b4_ << 4) ^ (8'h07 & (b5 >> 4)) ^ (8'h0B & b5) ^ (8'h0D & (b4 >> 4)) ^ (8'h0E &
b4);

b5 = b5_;

b4 = b4_;

b3_ = (8'h0B & (b3 >> 4)) ^ (8'h0D & b3) ^ (8'h0E & (b2 >> 4)) ^ (8'h07 & b2);

b3_ = (b3_ << 4) ^ (8'h0D & (b3 >> 4)) ^ (8'h0E & b3) ^ (8'h07 & (b2 >> 4)) ^ (8'h0B &
b2);

b2_ = (8'h0E & (b3 >> 4)) ^ (8'h07 & b3) ^ (8'h0B & (b2 >> 4)) ^ (8'h0D & b2);
```

```
                b2_ = (b2_ << 4) ^ (8'h07 & (b3 >> 4)) ^ (8'h0B & b3) ^ (8'h0D & (b2 >> 4)) ^ (8'h0E &
b2);

                b3 = b3_;

                b2 = b2_;

                b1_ = (8'h07 & (b1 >> 4)) ^ (8'h0B & b1) ^ (8'h0D & (b0 >> 4)) ^ (8'h0E & b0);

                b1_ = (b1_ << 4) ^ (8'h0B & (b1 >> 4)) ^ (8'h0D & b1) ^ (8'h0E & (b0 >> 4)) ^ (8'h07 &
b0);

                b0_ = (8'h0D & (b1 >> 4)) ^ (8'h0E & b1) ^ (8'h07 & (b0 >> 4)) ^ (8'h0B & b0);

                b0_ = (b0_ << 4) ^ (8'h0E & (b1 >> 4)) ^ (8'h07 & b1) ^ (8'h0B & (b0 >> 4)) ^ (8'h0D &
b0);

                b1 = b1_;

                b0 = b0_;

                block0Next = {b0,b1,b2,b3};

                block1Next = {b4,b5,b6,b7};

                result_data = block0Next;

                result_addr = instruction_word_in[27:14];

                SRCounterNext = 1;

        end

        else if (SRCounterCurrent == 1) begin

                result_data = block1Current;

                result_addr = instruction_word_in[13:0];

                SRCounterNext = 0;

        end

        result_wr_en = 1;

end

`S01: begin
```

```
        b0 = operand_a_actual_data[31:24];
        b1 = operand_a_actual_data[23:16];
        b2 = operand_a_actual_data[15:8];
        b3 = operand_a_actual_data[7:0];
        if ( instruction_word_in[13:0] == 0 )begin
                b0r = (  s0[ ( 63 - ( (b0 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b0 & 8'h0F) * 4 )  -: 4 ] );
                b1r = (  s0[ ( 63 - ( (b1 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b1 & 8'h0F) * 4 )  -: 4 ] );
                b2r = (  s0[ ( 63 - ( (b2 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b2 & 8'h0F) * 4 )  -: 4 ] );
                b3r = (  s0[ ( 63 - ( (b3 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b3 & 8'h0F) * 4 )  -: 4 ] );
        end
        else if( instruction_word_in[13:0] == 1 )begin
                b0r = (  s1[ ( 63 - ( (b0 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b0 & 8'h0F) * 4 )  -: 4 ] );
                b1r = (  s1[ ( 63 - ( (b1 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b1 & 8'h0F) * 4 )  -: 4 ] );
                b2r = (  s1[ ( 63 - ( (b2 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b2 & 8'h0F) * 4 )  -: 4 ] );
                b3r = (  s1[ ( 63 - ( (b3 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b3 & 8'h0F) * 4 )  -: 4 ] );
        end
        result_wr_en = 1;
        result_addr = instruction_word_in[27:14];
        result_data = {b0r,b1r,b2r,b3r};
end
`XOR: begin
        result_wr_en = 1;
        result_addr = instruction_word_in[27:14];
        result_data = (operand_a_actual_data ^ operand_b_actual_data);
```

```
end

`RKPRIME: begin

        if(RKPRIMECounterCurrent == 0)begin

                b0 = operand_a_actual_data[31:24];

                b1 = operand_a_actual_data[23:16];

                b2 = operand_a_actual_data[15:8];

                b3 = operand_a_actual_data[7:0];

                b4 = operand_b_actual_data[31:24];

                b5 = operand_b_actual_data[23:16];

                b6 = operand_b_actual_data[15:8] ;

                b7 = operand_b_actual_data[7:0]  ;

                b0r = ((b1 << 7) & 8'h80) ^ (b0 >> 1);

                b1r = ((b2 << 7) & 8'h80) ^ (b1 >> 1);

                b2r = ((b3 << 7) & 8'h80) ^ (b2 >> 1);

                b3r = ((b4 << 7) & 8'h80) ^ (b3 >> 1);

                b4r = ((b5 << 7) & 8'h80) ^ (b4 >> 1);

                b5r = ((b6 << 7) & 8'h80) ^ (b5 >> 1);

                b6r = ((b7 << 7) & 8'h80) ^ (b6 >> 1);

                b7r = ((b0 << 7) & 8'h80) ^ (b7 >> 1);

                b0r = b0r ^ ((b7 >> 7) & 8'h01);

                block0Next = {b0r,b1r,b2r,b3r};

                block1Next = {b4r,b5r,b6r,b7r};

                result_addr = instruction_word_in[13:0] + 1;

                RKPRIMECounterNext = 1;

                result_data = block0Next;
```

```
end
else if(RKPRIMECounterCurrent == 1)begin
        RKPRIMECounterNext = 0;
        result_data = block1Current;
        result_addr = instruction_word_in[13:0] + 2;
end
result_wr_en = 1;
```

APPENDIX C: THE RTL FOR 64-BIT CUSTOMIZED VSCPU

```
`SR: begin
        b0 = operand_a_actual_data[63:56];
        b1 = operand_a_actual_data[55:48];
        b2 = operand_a_actual_data[47:40];
        b3 = operand_a_actual_data[39:32];
        b4 = operand_a_actual_data[31:24];
        b5 = operand_a_actual_data[23:16];
        b6 = operand_a_actual_data[15:8] ;
        b7 = operand_a_actual_data[7:0]  ;
        if ( instruction_word_in[4:0] == 0 )begin
                b7_ = b7;
                b7 = (b7 & 8'hF0) ^ (b5 & 8'h0F);
                b5 = (b5 & 8'hF0) ^ (b3 & 8'h0F);
                b3 = (b3 & 8'hF0) ^ (b1 & 8'h0F);
                b1 = (b1 & 8'hF0) ^ (b7_ & 8'h0F);
                b0_ = b0;
                b2_ = b2;
                b0 = (b4 & 8'hF0) ^ (b2 & 8'h0F);
                b2 = (b6 & 8'hF0) ^ (b4 & 8'h0F);
                b4 = (b0_ & 8'hF0) ^ (b6 & 8'h0F);
                b6 = (b2_ & 8'hF0) ^ (b0_ & 8'h0F);
        end
        else if( instruction_word_in[4:0] == 1 )begin
                b1_ = b1;
```

```
                b1 = (b1 & 8'hF0) ^ (b3 & 8'h0F);
                b3 = (b3 & 8'hF0) ^ (b5 & 8'h0F);
                b5 = (b5 & 8'hF0) ^ (b7 & 8'h0F);
                b7 = (b7 & 8'hF0) ^ (b1_ & 8'h0F);
                b6_ = b6;
                b4_ = b4;
                b6 = (b2 & 8'hF0) ^ (b4 & 8'h0F);
                b4 = (b0 & 8'hF0) ^ (b2 & 8'h0F);
                b2 = (b6_ & 8'hF0) ^ (b0 & 8'h0F);
                b0 = (b4_ & 8'hF0) ^ (b6_ & 8'h0F);
        end
        result_wr_en = 1;
        result_addr = instruction_word_in[9:5];
        result_data = {b0,b1,b2,b3,b4,b5,b6,b7};
end
`M01: begin
        b0 = operand_a_actual_data[63:56];
        b1 = operand_a_actual_data[55:48];
        b2 = operand_a_actual_data[47:40];
        b3 = operand_a_actual_data[39:32];
        b4 = operand_a_actual_data[31:24];
        b5 = operand_a_actual_data[23:16];
        b6 = operand_a_actual_data[15:8] ;
        b7 = operand_a_actual_data[7:0]  ;
        b7_ = (8'h07 & (b7 >> 4)) ^ (8'h0B & b7) ^ (8'h0D & (b6 >> 4)) ^ (8'h0E & b6);
```

```
b7_ = (b7_ << 4) ^ (8'h0B & (b7 >> 4)) ^ (8'h0D & b7) ^ (8'h0E & (b6 >> 4)) ^ (8'h07 & b6);

b6_ = (8'h0D & (b7 >> 4)) ^ (8'h0E & b7) ^ (8'h07 & (b6 >> 4)) ^ (8'h0B & b6);

b6_ = (b6_ << 4) ^ (8'h0E & (b7 >> 4)) ^ (8'h07 & b7) ^ (8'h0B & (b6 >> 4)) ^ (8'h0D & b6);

b7 = b7_;

b6 = b6_;

b5_ = (8'h0B & (b5 >> 4)) ^ (8'h0D & b5) ^ (8'h0E & (b4 >> 4)) ^ (8'h07 & b4);

b5_ = (b5_ << 4) ^ (8'h0D & (b5 >> 4)) ^ (8'h0E & b5) ^ (8'h07 & (b4 >> 4)) ^ (8'h0B & b4);

b4_ = (8'h0E & (b5 >> 4)) ^ (8'h07 & b5) ^ (8'h0B & (b4 >> 4)) ^ (8'h0D & b4);

b4_ = (b4_ << 4) ^ (8'h07 & (b5 >> 4)) ^ (8'h0B & b5) ^ (8'h0D & (b4 >> 4)) ^ (8'h0E & b4);

b5 = b5_;

b4 = b4_;

b3_ = (8'h0B & (b3 >> 4)) ^ (8'h0D & b3) ^ (8'h0E & (b2 >> 4)) ^ (8'h07 & b2);

b3_ = (b3_ << 4) ^ (8'h0D & (b3 >> 4)) ^ (8'h0E & b3) ^ (8'h07 & (b2 >> 4)) ^ (8'h0B & b2);

b2_ = (8'h0E & (b3 >> 4)) ^ (8'h07 & b3) ^ (8'h0B & (b2 >> 4)) ^ (8'h0D & b2);

b2_ = (b2_ << 4) ^ (8'h07 & (b3 >> 4)) ^ (8'h0B & b3) ^ (8'h0D & (b2 >> 4)) ^ (8'h0E & b2);

b3 = b3_;

b2 = b2_;

b1_ = (8'h07 & (b1 >> 4)) ^ (8'h0B & b1) ^ (8'h0D & (b0 >> 4)) ^ (8'h0E & b0);

b1_ = (b1_ << 4) ^ (8'h0B & (b1 >> 4)) ^ (8'h0D & b1) ^ (8'h0E & (b0 >> 4)) ^ (8'h07 & b0);


b0_ = (8'h0D & (b1 >> 4)) ^ (8'h0E & b1) ^ (8'h07 & (b0 >> 4)) ^ (8'h0B & b0);

b0_ = (b0_ << 4) ^ (8'h0E & (b1 >> 4)) ^ (8'h07 & b1) ^ (8'h0B & (b0 >> 4)) ^ (8'h0D & b0);

b1 = b1_;

b0 = b0_;

result_wr_en = 1;

result_addr = instruction_word_in[9:5];
```

```
        result_data = {b0,b1,b2,b3,b4,b5,b6,b7};
end
`S01: begin

        b0 = operand_a_actual_data[63:56];
        b1 = operand_a_actual_data[55:48];
        b2 = operand_a_actual_data[47:40];
        b3 = operand_a_actual_data[39:32];
        b4 = operand_a_actual_data[31:24];
        b5 = operand_a_actual_data[23:16];
        b6 = operand_a_actual_data[15:8] ;
        b7 = operand_a_actual_data[7:0]  ;
        if ( instruction_word_in[4:0] == 0 )begin
                b0r = (  s0[ ( 63 - ( (b0 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b0 & 8'h0F) * 4 )  -: 4 ] );
                b1r = (  s0[ ( 63 - ( (b1 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b1 & 8'h0F) * 4 )  -: 4 ] );
                b2r = (  s0[ ( 63 - ( (b2 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b2 & 8'h0F) * 4 )  -: 4 ] );
                b3r = (  s0[ ( 63 - ( (b3 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b3 & 8'h0F) * 4 )  -: 4 ] );
                b4r = (  s0[ ( 63 - ( (b4 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b4 & 8'h0F) * 4 )  -: 4 ] );
                b5r = (  s0[ ( 63 - ( (b5 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b5 & 8'h0F) * 4 )  -: 4 ] );
                b6r = (  s0[ ( 63 - ( (b6 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b6 & 8'h0F) * 4 )  -: 4 ] );
                b7r = (  s0[ ( 63 - ( (b7 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s0[ 63 - ((b7 & 8'h0F) * 4 )  -: 4 ] );
        end
        else if( instruction_word_in[4:0] == 1 )begin

                b0r = (  s1[ ( 63 - ( (b0 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b0 & 8'h0F) * 4 )  -: 4 ] );
```

```
                b1r = (  s1[ ( 63 - ( (b1 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b1 & 8'h0F) * 4 )  -: 4 ] );

                b2r = (  s1[ ( 63 - ( (b2 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b2 & 8'h0F) * 4 )  -: 4 ] );

                b3r = (  s1[ ( 63 - ( (b3 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b3 & 8'h0F) * 4 )  -: 4 ] );

                b4r = (  s1[ ( 63 - ( (b4 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b4 & 8'h0F) * 4 )  -: 4 ] );

                b5r = (  s1[ ( 63 - ( (b5 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b5 & 8'h0F) * 4 )  -: 4 ] );

                b6r = (  s1[ ( 63 - ( (b6 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b6 & 8'h0F) * 4 )  -: 4 ] );

                b7r = (  s1[ ( 63 - ( (b7 >> 4) * 4 ) ) -: 4 ] << 4) ^ ( s1[ 63 - ((b7 & 8'h0F) * 4 )  -: 4 ] );
        end
        result_wr_en = 1;
        result_addr = instruction_word_in[9:5];
        result_data = {b0r,b1r,b2r,b3r,b4r,b5r,b6r,b7r};
end
`XOR: begin
        result_wr_en = 1;
        result_addr = instruction_word_in[9:5];
        result_data = (operand_a_actual_data ^ operand_b_actual_data);
end
`RKPRIME: begin
        b0 = operand_a_actual_data[63:56];
        b1 = operand_a_actual_data[55:48];
        b2 = operand_a_actual_data[47:40];
        b3 = operand_a_actual_data[39:32];
        b4 = operand_a_actual_data[31:24];
        b5 = operand_a_actual_data[23:16];
        b6 = operand_a_actual_data[15:8] ;
```

```
        b7 = operand_a_actual_data[7:0]  ;

        b0r = ((b1 << 7) & 8'h80) ^ (b0 >> 1);

        b1r = ((b2 << 7) & 8'h80) ^ (b1 >> 1);

        b2r = ((b3 << 7) & 8'h80) ^ (b2 >> 1);

        b3r = ((b4 << 7) & 8'h80) ^ (b3 >> 1);

        b4r = ((b5 << 7) & 8'h80) ^ (b4 >> 1);

        b5r = ((b6 << 7) & 8'h80) ^ (b5 >> 1);

        b6r = ((b7 << 7) & 8'h80) ^ (b6 >> 1);

        b7r = ((b0 << 7) & 8'h80) ^ (b7 >> 1);

        b0r = b0r ^ ((b7 >> 7) & 8'h01);

        result_wr_en = 1;

        result_addr = instruction_word_in[4:0];

        result_data = {b0r,b1r,b2r,b3r,b4r,b5r,b6r,b7r};

end
```