2015•2016
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: elektronica-ICT

Masterproef
Evaluation of CAESAR candidates on FPGA

Promotor :
Prof. dr. ir. Nele MENTENS

Promotor :
dr. BEGÜL BILGIN

Copromotor :
Dhr. BOHAN YANG
Dhr. DANILO SIJACIC

Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

Jasper Gorissen
Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

KU LEUVEN | universiteit hasselt

KU LEUVEN | universiteit hasselt

2015•2016
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterproef
Evaluation of CAESAR candidates on FPGA

Promotor :
Prof. dr. ir. Nele MENTENS

Promotor :
dr. BEGÜL BILGIN

Copromotor :
Dhr. BOHAN YANG
Dhr. DANILO SIJACIC

Jasper Gorissen

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

universiteit hasselt | KU LEUVEN

Acknowledgements

First off all, I would like to express my greatest appreciation to all my supervisors, Prof. dr. Ir. Nele Mentens, dr. Begül Bilgin, Mr. Bohan Yang and Mr. Danilo Šijačić , for their encouragement, great advice and enthusiasm in guiding me in the course of this thesis. Prof. dr. Ir. Mentens initial advice helped me get started, and her continued help on some of the more difficult mathematical equations helped me overcome many problems. Dr. Bilgin and Mr. Yang both helped immensely in understanding the authenticated encryption algorithms and more than once offered vital advice when I was stuck. They were always available for questions and also offered many remarks on the thesis, which helped to improve it. Mr. Šijačić offered a great and patient step by step tutorial on the Design compiler, which allowed me to write a script to automate the generation of results. He also gave great advice on this thesis and on the choice of comparison metrics.

I would also like to thank my mother and her friend. They supported my decisions and provided me with everything to get me where I am now. I also would like to thank my brother, friends, and close family for their support. A special thanks also goes out to my girlfriend, who kept my motivation high, and who helped even when she had little time.

Table of contents

Acknowledgements i
List of tables v
List of figures vii
List of equations ix
List of graphs xi
List of abbreviations xiii
Abstract (Nederlands) xv
Abstract xvii

1 Introduction 1
2 Materials and methods 3
2.1 Authenticated encryption 3
2.2 Used programs 5
2.2.1 Xilinx ISE WebPACK 5
2.2.2 AEAD API 6
2.2.3 ATHENa & Design Compiler 9
3 Trivia-ck 11
3.1 Algorithm 11
3.1.1 Trivia –SC 12
3.1.2 VPV-Hash 13
3.2 VHDL code 14
3.2.1 Top level block 14
3.2.2 Data Path 15
3.2.3 Finite State Machine 17
3.3 Trivia-ck optimisation 18
3.3.1 Basic code 19
3.3.2 Area optimisations 19
3.3.3 Speed optimisations 23
3.3.4 Trivia-ck version 2 26
3.3.5 Results 26
4 Ketje 33
4.1 Algorithm 33
4.1.1 Sponge construction 33
4.1.2 MonkeyWrap construction 34

4.1.3 MonkeyDuplex construction 35
4.1.4 KECCAK-p permutation 36
4.2 VHDL code 40
4.2.1 Data path 40
4.2.2 State machine 42
4.3 Ketje optimisation 42
4.3.1 Area optimisation 42
4.3.2 Speed optimisation 46
4.3.3 State machine optimisations 48
4.3.4 Results 51
5 MORUS 55
5.1 Algorithm 55
5.1.1 State update 57
5.2 VHDL code 59
5.2.1 Data path 59
5.2.2 State machine 60
5.3 MORUS optimisation 61
5.3.1 Results 62
6 Performance comparison 65
7 Conclusion & discussion 69

Literature 71
Appendix 73

List of tables

Table 2-1: CAESAR candidates 4
Table 3-1: Estimated gate counts of 1-bit to 64-bit hardware implementations [18] 12
Table 3-2: Area results for basic Trivia-ck using AEAD wrapper 26
Table 3-3: Area usage for basic Trivia-ck without AEAD wrapper 26
Table 3-4: Area results for area optimised Trivia-ck 27
Table 3-5: Area results for speed optimised Trivia-ck 27
Table 3-6: Area results for Trivia-ck v2 27
Table 3-7: Timing results for basic Trivia-ck 27
Table 3-8: Timing results for area optimised Trivia-ck 28
Table 3-9: Timing results for speed optimised Trivia-ck 28
Table 3-10: Timing results for Trivia-ck v2 28
Table 3-11: ASIC results for basic Trivia-ck code 31
Table 3-12: ASIC results for Trivia_ck area optimisation 31
Table 3-13: ASIC results for Trivia_ck speed optimisation 31
Table 3-14: ASIC results for Trivia_ck v2 31
Table 4-1: Ketje parameters 33
Table 4-2: Rho offsets [24] 38
Table 4-3: Round constants for each of the modes in MonkeyDuplex 40
Table 4-4: Area results for speed optimised KetjeJr without/with optimised state machine 51
Table 4-5: Area results for area optimised KetjeJr without/with optimised state machine 51
Table 4-6: Area results for speed optimised KetjeSr without/with optimised state machine 51
Table 4-7: Area results for area optimised KetjeSr with optimised state machine 51
Table 4-8: Timing results for speed optimised KetjeJr without/with optimised state machine 52
Table 4-9: Timing results for area optimised KetjeJr without/with optimised state machine 52
Table 4-10: Timing results for speed optimised KetjeSr without/with optimised state machine 52
Table 4-11: Timing results for area optimised KetjeSr with optimised state machine 52
Table 4-12: Latency results for Ketje 53
Table 4-13: ASIC results for Ketje implementations with optimised state machine 54
Table 5-1: MORUS parameters [26] 55
Table 5-2: Rotation constants of MORUS [26] 57
Table 5-3: Shift constants of MORUS [26] 57
Table 5-4: Area results for MORUS-640. 62
Table 5-5: Area results for MORUS-1280. 62
Table 5-6: Timing results for MORUS-640 63
Table 5-7: Timing results for MORUS-1280 63
Table 5-8: ASIC results for MORUS implementations 63
Table 6-1: Results for minimum area 65
Table 6-2: Results for maximum speed 65
Table 6-3: Throughput/ area results for both optimisation paths 66
Table 6-4: CAESAR results implemented with the AEAD core on Virtex 6 [29] 67

List of figures

Figure 2-1: EtM, E&M, MtE [11] 3
Figure 2-2: Generic stream cipher [13] 4
Figure 2-3: The sponge construction [14] 5
Figure 2-4: An ISim testbench output 5
Figure 2-5: AEAD interface [15] 6
Figure 2-6: Two formats for SDI [15] 7
Figure 2-7: Formats for PDI [15] 7
Figure 2-8: The segment header [15] 8
Figure 2-9: The AEAD interface [15] 8
Figure 3-1: Trivia-ck circuit [17] 11
Figure 3-2: The Trivia-SC cipherstream and the Trivium cipherstream [17], [19] 13
Figure 3-3: Trivia-ck ciphercore 15
Figure 3-4: Trivia-ck data path 15
Figure 3-5: VPVhash [17] 16
Figure 3-6: Field multiplier 17
Figure 3-7: Trivia state machine 18
Figure 3-8: Testbench results from the basic Trivia-ck version 19
Figure 3-9: A 4:1 mux using 2 LUT's [20] 20
Figure 3-10: state machine for the area optimisation 22
Figure 3-11: Pipelined field multiplier 23
Figure 3-12: state machine of the speed optimised code 25
Figure 4-1: The sponge construction [14] 33
Figure 4-2 : Duplex construction [14] 34
Figure 4-3: Encrypting data using MonkeyWrap [23] 35
Figure 4-4: The MonkeyDuplex construction [23] 36
Figure 4-5: Naming conventions for the KECCAK state [24] 37
Figure 4-6: Theta [24] 37
Figure 4-7: The rho step [24] 38
Figure 4-8: The pi step [24] 39
Figure 4-9: The chi step [24] 39
Figure 4-10: Ketje data path 41
Figure 4-11: Area optimised KECCAK round block [25] 43
Figure 4-12: Ketje area state machine 45
Figure 4-13: Ketje speed state machine 47
Figure 4-14: Area optimised Ketje with less states. 49
Figure 4-15: Speed optimised Ketje with less states 50
Figure 5-1: The encryption process of MORUS 56
Figure 5-2: The state update function [26] 58
Figure 5-3: MORUS data path 59
Figure 5-4: MORUS state machine 61

List of equations

Equation 3-1: Vandermonde Matrix [17] 13
Equation 3-2: Primitive polynomials of GF(2^{32}) and GF(2^{64}) [17] 14
Equation 3-3: Extended input generation [17] 14
Equation 3-4: Throughput calculation [22] 28
Equation 3-5: Throughput calculation for short messages [22] 29
Equation 3-6 : Delay calculation in cycles 29
Equation 3-7: Latency 30
Equation 4-1: Input string to 3d matrix mapping [23] 36
Equation 4-2: Round constant value [23] 39
Equation 4-3: rc constant value [23] 39

List of graphs

Graph 6-1: Throughput/ Area of CAESAR candidates (low Throughput) 67
Graph 6-2: Throughput/ Area of CAESAR candidates (high throughput)... 68

List of abbreviations

AD	Associated data
AE	Authenticated encryption
AEAD	Authenticated encryption with authenticated data
AES	Advanced encryption standard
API	Application programming interface
ASIC	Application-specific integrated circuit
ATHENa	Automated tool for hardware evaluation
CAESAR	Competition for authenticated encryption: security, applicability, and robustness
COSIC	Computer security and industrial cryptography
DES	Data encryption standard
DO	Data output
E&M	Encrypt-and-MAC
ECCode	Error correcting code
EOI	End of input
EOT	End of type
EtM	Encrypt-then-MAC
FF	Flip-flop
FM	Field multiplier
FPGA	Field programmable gate array
FSM	Finite state machine
GF	Galois field
HDL	Hardware description language
I/O	Input/output
ISE	Integrated synthesis environment
IV	Initialization vector
LSB	Least significant bit
LUT	Lookup table
MAC	Message authentication code
MSB	Most significant bit
MtE	MAC-then-Encrypt
Npub	Public message number
Nsec	Secret message number
PAR	Place and route
PDI	Public data input
RC	Round constant
SDI	Secret data input
SHA-3	Secure hash algorithm 3
VHDL	Very high speed integrated circuit hardware description language
WEP	Wired equivalent privacy
XOR	Exclusive or

Abstract (Nederlands)

Evaluation of CAESAR candidates on FPGA

Auteur: Jasper Gorissen

Interne promotor: Prof. dr. ir. Nele MENTENS

Externe promotor: Dr. Begül Bilgin

Externe copromotoren: Dhr. Bohan Yang & Dhr. Danilo Šijačić

Het beveiligen van data is essentieel in ons leven. Omdat krakers manieren vinden om bestaande cryptografische algoritmen te kraken, moeten deze evolueren. Authenticated encryption (AE) biedt zowel confidentiality als authenticity van data simultaan aan. De nood voor AE is er gekomen nadat het combineren van bestaande algoritmen die confidentiality aanboden met die wat authenticity aanboden resulteerde in onveilige schema's. Nieuwe AE algoritmen worden ingezonden naar de CAESAR cryptografische wedstrijd. Onder andere, het departement Elektrotechniek (ESAT) van KU Leuven test deze algoritmen op sterkte en hardware prestaties om zo hun kwaliteit te bepalen.

Het doel van deze masterproef is het implementeren van drie CAESAR inzendingen: Trivia-ck, Ketje en MORUS in de AEAD API. Twee prestatie-eigenschappen die de mogelijke applicaties helpen bepalen, oppervlakte en snelheid, zijn gekozen als optimalisatiedoelwitten voor deze inzendingen. De implementaties zijn getest met ATHENa en Design Compiler.

De verschillende implementaties tonen de minimale oppervlakte en maximale snelheid van elk algoritme, alsook de afwegingen tussen de twee. De resultaten zijn ook vergeleken met andere CAESAR inzendingen, om zo een beter inzicht te krijgen in hun kwaliteit. Hier is aangetoond dat Ketje het kleinste oppervlakte heeft en dat MORUS een uitstekende snelheid/oppervlakte verhouding heeft. Trivia-ck, hoewel deze een hoge throughput heeft in de snelheid geoptimaliseerde versie, presteert niet goed in oppervlakte vergeleken met de rest.

Abstract

Evaluation of CAESAR candidates on FPGA

Author: Jasper Gorissen

Internal supervisor: Prof. dr. ir. Nele MENTENS

External supervisor: Dr. Begül Bilgin

External co-supervisors: Dhr. Bohan Yang & Dhr. Danilo Šijačić

Data security is an essential part of our daily lives. However, cryptanalysts discover ways to crack existing cryptographic algorithms, making it necessary to improve them. Authenticated encryption (AE) offers the confidentiality and authenticity of data simultaneously. The need for AE has risen because combining existing algorithms that provide confidentiality with those that provide authenticity results in insecure schemes. New AE algorithms are submitted to the CAESAR cryptographic competition. Amongst others, the department of Electrical Engineering (ESAT) at KU Leuven tests the AE algorithms on their strength and performance to help determine their overall quality.

The goal of this thesis is the implementation of three CAESAR submissions: Trivia-ck, Ketje and MORUS, in VHDL, using the AEAD API. Two performance metrics, area and speed, which help determine the range of possible applications, are chosen as optimisation targets for these submissions. The implementations are tested on those metrics using ATHENa and Design Compiler.

The different implementations show the minimal area and the maximal speed of each algorithm, as well as the trade-offs between the metrics. The results are also compared to other CAESAR submissions, in order to get a better perspective on the quality of the implementations. Here it shows that Ketje has the smallest area usage and MORUS has an excellent speed/area ratio. Trivia-ck, while having good throughput in the speed optimised version, underperforms in area usage compared to others.

1 Introduction

Cryptography is the field of research that encompasses the practice and study of techniques used to secure communication in the presence of unwanted third parties, which are called adversaries [1]. Cryptographic algorithms are used in many parts of our lives, from securing electronic payments and cell phones to internet security and many more. Algorithms used to encrypt and decrypt data are called ciphers. Encryption is the transformation of information from a useful form of understanding to an opaque form of understanding, and decryption is the opposite. This transformation of information is usually achieved by using a key. In symmetric key algorithms, data is encrypted using a secret key, and only those with the same secret key can decrypt the data. In the case of asymmetric key algorithms, two keys are used, one public and one secret, that are somehow paired together. The sender encrypts the data using the receiver public key, and now only the receiver can decrypt this data using his secret key.

The common goals in cryptography are message confidentiality, integrity and sender authentication [2]. Confidentiality is the privacy of the message, which is achieved by using a key as mentioned before. Message integrity (the ability to detect changes in the message content) and authenticity (being able to verify the identity of the sender) can be achieved by using a message authentication code (MAC). In order to achieve confidentiality and authentication at the same time, authenticated encryption algorithms are used. The need for this type of encryption has arisen from observations that combining secure encryption schemes (to provide confidentiality) with secure MACs (to provide authenticity) was difficult and prone to errors [3].

The other aspect of cryptology is cryptanalysis, which studies the methods on how to crack ciphers and their implementations. Since cryptanalysts discover new cryptanalytic attacks and have more powerful tools at their disposal to crack existing algorithms, cryptographic algorithms need to evolve. For instance, the data encryption standard (DES), the once predominant symmetric-key algorithm, was cracked by discovering the key in 22 hours and 15 minutes using brute force attacks, its weakness being its small 56-bit size [4]. In order to promote the creation of new algorithms, cryptographic competitions are organised. For instance, the advanced encryption standard (AES) competition was announced in 1997, called for "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century" [5]. The winning algorithm, Rijndael, has superseded DES and is now used almost everywhere in secure applications, like bankcards and network security [5]. AES was followed by other competitions, like eSTREAM and SHA-3, each with their own focus. The latest of these competitions, CAESAR, focuses on authenticated encryption. CAESAR stands for ‘Competition for Authenticated Encryption: Security, Applicability, and Robustness’ [5].

In the competition, the submitted algorithms are tested against several cryptanalytic attacks to measure their strength as well on their performance, such as area usage, speed and power usage in hardware. These performance metrics help determine the range of possible applications. The smaller the hardware usage, the smaller the needed chip, the bigger the range of applications. The same goes for speed and power. An ideal algorithm will be secure, small and fast. The submissions are also tested on their performance in software.

The goal of this thesis is testing the area and speed hardware performance of several submitted CAESAR candidates. The chosen algorithms are Trivia-ck, Ketje and MORUS. They are tested on field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) chips. The difference between FPGA and ASIC is that FPGAs are premade chips, with a fixed number of logic

cells, depending on the chip, grouped together in slices, whereas in ASIC chips the number of logic gates is determined by the design. FPGA's are reprogrammable and are often used for prototyping. After the prototyping stage is done, developers usually switch over to ASIC, where they can control the area, timing etc.

The submitted algorithms are written in C-code or pseudocode, which can't be used to evaluate hardware. So this hardware code needs to be created, using a Hardware Description Language (HDL) such as VHDL or Verilog [6]. The objective of this thesis is to investigate the minimal area usage and maximal speed of each algorithm, and investigating the trade-offs between these two metrics. In order to achieve this, two versions of the code are created, one that is optimised for speed and another for area. Two different implementations are needed because most optimisations used to improve one characteristic will be at cost of the other. This code then needs to be embedded in an application programming interface (API), which allows the outside world to interact with the algorithm. The results on area and speed are then used in a comparative study to determine the quality of the algorithms regarding these metrics.

This thesis is done in cooperation with ESAT. ESAT is the department of Electrical Engineering of KU Leuven that does research on international level and offers academic level education on the subjects of electronics, electrical engineering and information processing. The department also explores a wide array of technological innovations in the field of energy, integrated circuits, image and voice processing and telecommunication systems [7].

COSIC, which stands for Computer Security and Industrial Cryptography, is a research group branched of ESAT that specializes in digital security. Their studies are applied in a broad range of domains, like electronic payments, communication, electronic ID, e-voting and the security of e-documents. Their research focus lies in the development of security architectures for information and communication systems, the building of security mechanisms for embedded systems and the design, evaluation and implementation of cryptographic algorithms and protocols [8].

In Section 2, the motivation for the chosen algorithms is given. The used programs are also discussed briefly. Sections 3, 4 and 5 offers a more in-depth look at the algorithms and their respective VHDL code and optimisations. In Section 6, the results are compared with one another and with other implementations and in Section 7, the conclusions are presented.

2 Materials and methods

2.1 Authenticated encryption

The concept of authenticity first appeared in the banking industry. Banks did not want to transmit their data and allow an attacker to flip bits undetected. Here the attacker would not need to decrypt the message, however flipping bits would make the encrypted message say “Post $800” instead of “Post $100” for instance [3]. At first, primitives that offer confidentiality and authenticity had been designed separately. Combining these two resulted in weak systems, such as the wired equivalent privacy (WEP), which used a cyclic redundancy check as hash function alongside a stream cipher for encryption. WEP keys could be cracked in minutes using simple hardware and computers [9]. So there was a need to find a way to combine a strong authentication mechanism with a strong cipher. Hugo Krawczyk published a paper in which he examined three commonly used combination methods [10]. These methods are EtM (encrypt then MAC), MtE (MAC then encrypt) and E&M (encrypt and MAC).

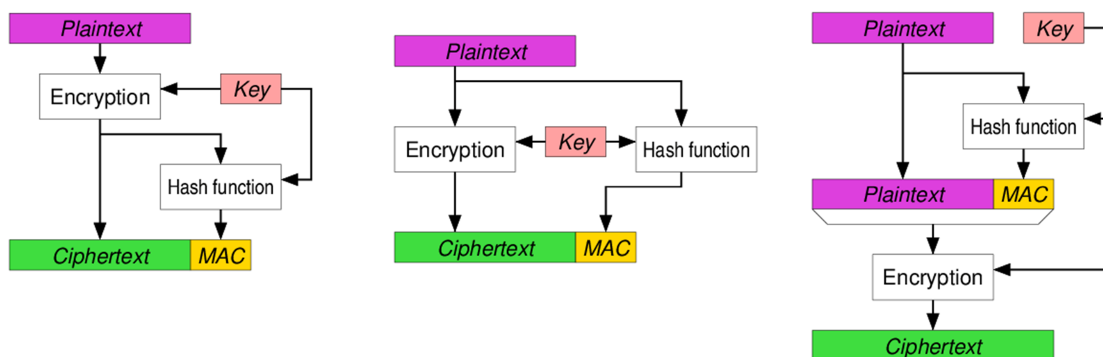


Figure 2-1: left: EtM, middle : E&M, right: MtE [11]

For their encryption input, these schemes use a key, message and optionally a header that will not be encrypted and instead is covered by authenticity protection. The output for encryption is the ciphertext and the tag or MAC. During decryption, the inputs are the ciphertext, tag and the optional header, which will then output the original message if the outputted tag is equal to the transmitted one.

In the beginning of the thesis, the choice was offered between the 30 CAESAR submissions that made it to the second round. The candidates that were chosen can be found in Table 2-1.

Table 2-1: CAESAR candidates

Cipher	Based on (type)	IV (bits)	Key (bits)	State (bits)
Trivia-ck	Trivium (Stream cipher)	128	128	385
Ketje	KECCAK-f (Sponge function)	80/128	96/128	200/400
MORUS	LRX (Stream cipher)	128	128/256	640/1280

Trivia-ck has been chosen because of previous experience with Trivium on which the cipher is based. Trivium is a stream cipher. This type of cipher combines its plaintext with a pseudorandom digit stream, one digit at a time. Stream ciphers usually have a smaller hardware footprint and are faster than block ciphers [12].

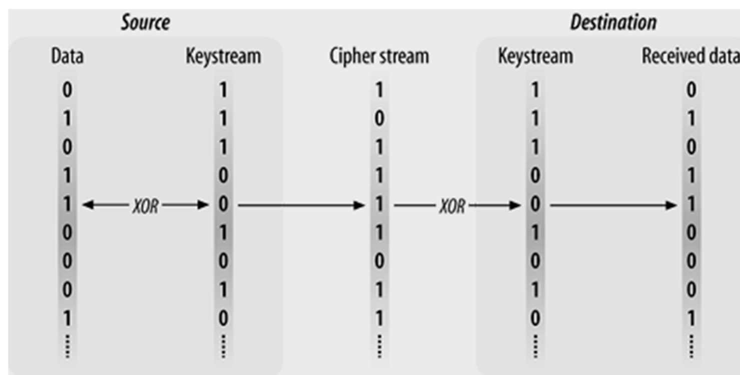


Figure 2-2: Generic stream cipher [13]

MORUS, the other AE algorithm based on a stream cipher, was chosen for its resemblance to Trivia-ck. Using this is interesting to see the differences in performance when comparing the two. It is also an algorithm built for speed, with 128/256 bits output per cycle and a fast state update function.

Ketje has been chosen since it shares the same permutation with KECCAK, of which a subset was chosen to be the new SHA-3 standard in 2007. Ketje is a sponge construction, which takes an input bit stream of any length to produce an output with a desired length in two distinct phases: absorbing, where the input is read into the state, and squeezing, where the state returns the output blocks. A more detailed explanation of the sponge construction is given in Section 4.

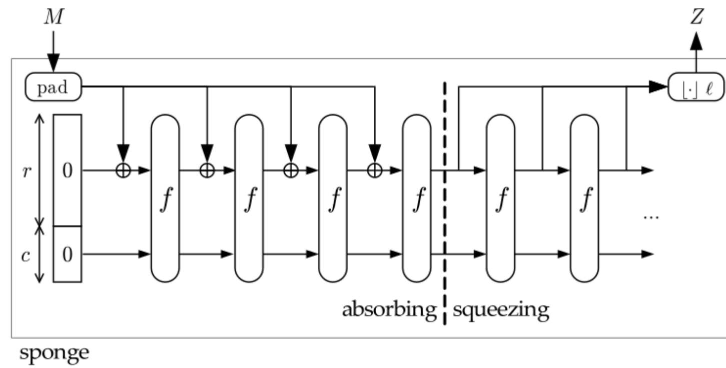


Figure 2-3: The sponge construction [14]

2.2 Used programs

During the course of the thesis, a number of programs are used for various purposes. These are: Xilinx integrated synthesis environment (ISE) WebPACK design software, the hardware Application Programming Interface (API) AEAD, ATHENa and Design Compiler. This section outlines them briefly.

2.2.1 Xilinx ISE WebPACK

The Xilinx ISE is a software tool used for the synthesis and analysis of hardware description languages (HDL) designed by Xilinx. It can be used to generate timing reports and area consumption on several Xilinx FPGA families, like the Spartan and Virtex FPGA's. Since the WebPACK is a free licence, not all of the mentioned families are available. It is mainly used in this thesis for its ISim simulator. Using ISim, the HDL code can be tested by simulating input signals and observing and verifying the output of the device under test.

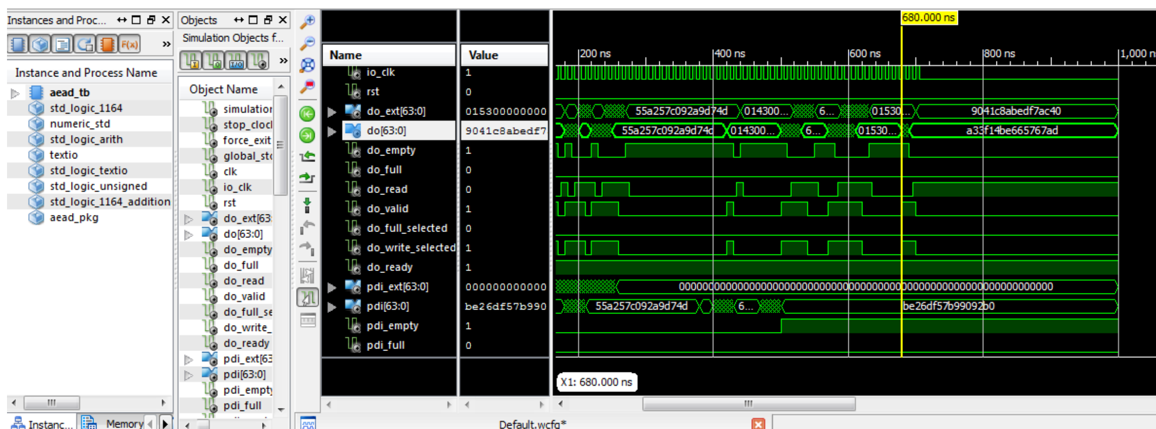


Figure 2-4: An ISim testbench output

As mentioned, the code is written in HDL, which is a specialized language to program digital logic circuits. There are two major HDLs: VHDL and Verilog. The one used in this thesis is VHDL because past experience with the language, but since the two languages are very similar, the code can be converted from one to another easily.

2.2.2 AEAD API

AEAD is a proposed universal hardware API for authenticated ciphers. In the call for CAESAR candidates, the software API has been clearly defined. The result is that software implementations can be compared equally. However, no similar hardware API has been proposed. As a result, hardware implementations use independently made API's, which can have a high influence on area and throughput, resulting in unreliable comparisons between the candidates [15]. The AEAD API offers several features like allowing input of an arbitrary size (a multiple of bytes only), a wide range of data port widths (between eight and 256), independent data and key inputs, support for encryption and decryption within the same core and more. The AEAD interface has three major data busses: one for public data (the associated data, the message, the public message number and tag), one for secret data (the key) and one for data outputs (the ciphertext and tag). Each of these busses have their own handshaking signals *valid*, which indicates that data is ready at the source, and *ready* which indicates that the destination is ready to receive. The interface is shown in Figure 2-5.

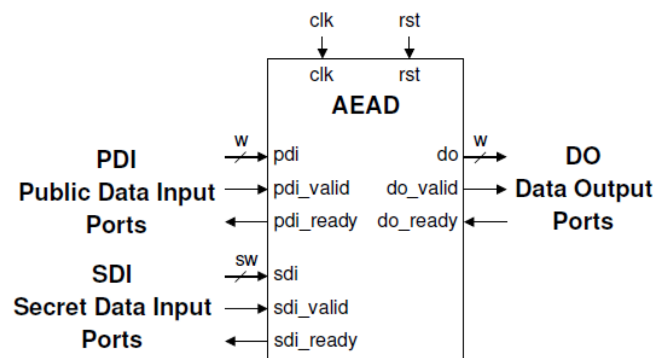


Figure 2-5: AEAD interface [15]

The format of the secret data input (SDI) starts with an instruction, which is either Load Key or Load Round Key, followed by the key or round key segments, whose size is determined by the width of the SDI. Each segment will have their own header.

Figure 2-6: Two formats for SDI [15]

The format for the public data input (PDI) is similar. It starts with an instruction: activate key (ACTKEY), authenticated encryption (ENC) or authenticated decryption (DEC). The activate key instruction, which is used to couple the secret key with this input, is called first, then either authenticated encryption or decryption. During encryption, the public data is segmented in the public message number (Npub), secret message number (Nsec), associated data (AD) and the message. During decryption, the segments are the same but include the ciphertext instead of the message, the encrypted secret message number instead of the Nsec, and the tag.

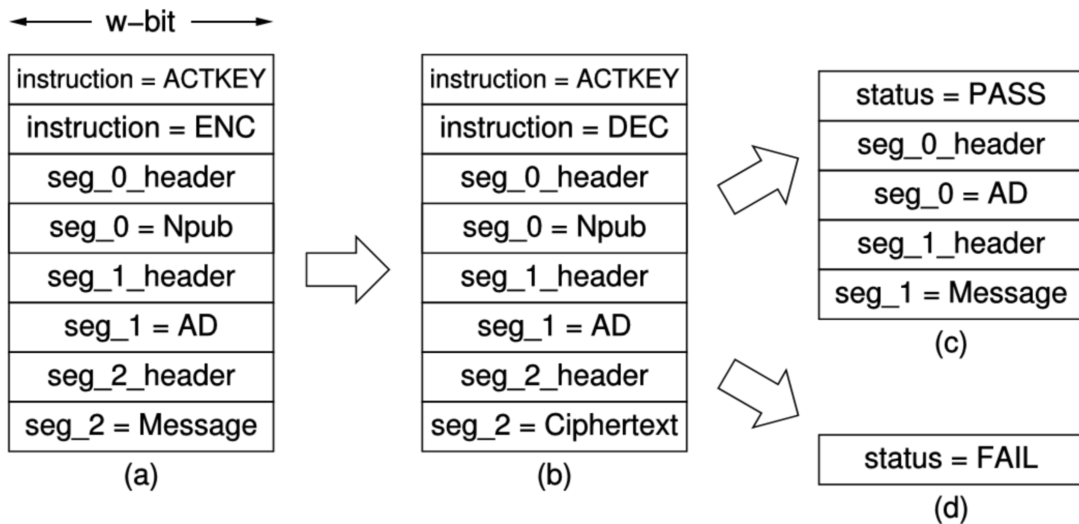


Figure 2-7: Formats for PDI. (a) is encryption, (b),(c) and (d) show decryption and the resulting pass or fail [15]

Each segment will have its own header that determines its type and length. It also notifies if it's the last of its type, or the last of the input.

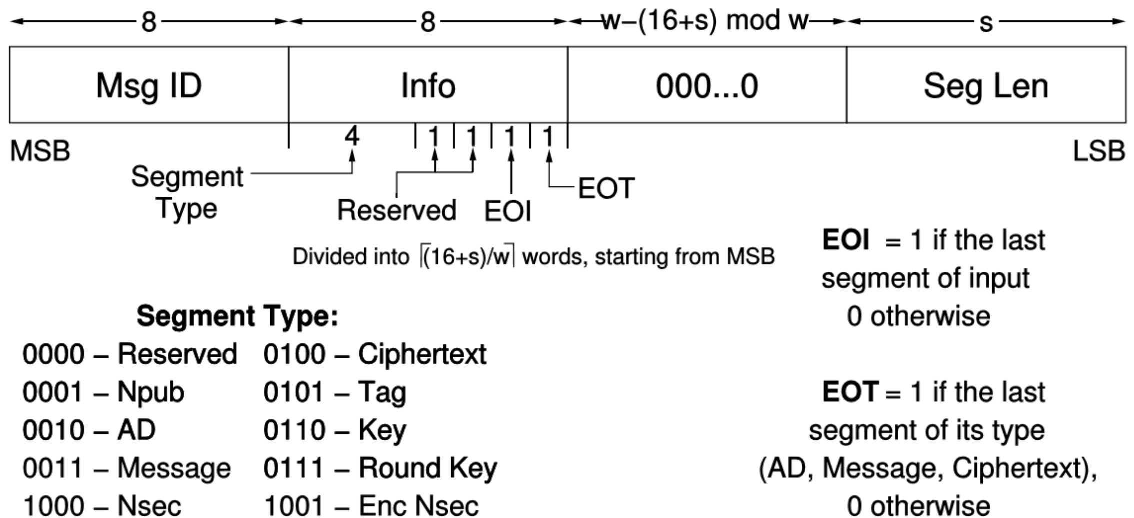


Figure 2-8: The segment header [15]

The AEAD interface is built up out of three parts: the pre-processor, the post-processor and the ciphercore. The pre-processor parses the segment headers, loads the keys and input blocks, pads the input blocks and keeps track of the number of data bytes left to process. The post-processor clears the padding out of the ciphertext or plaintext, converts the output blocks back into words and formats them into segments. During decryption, it stores the messages until the result of authentication is known. The ciphercore is the actual cipher algorithm.

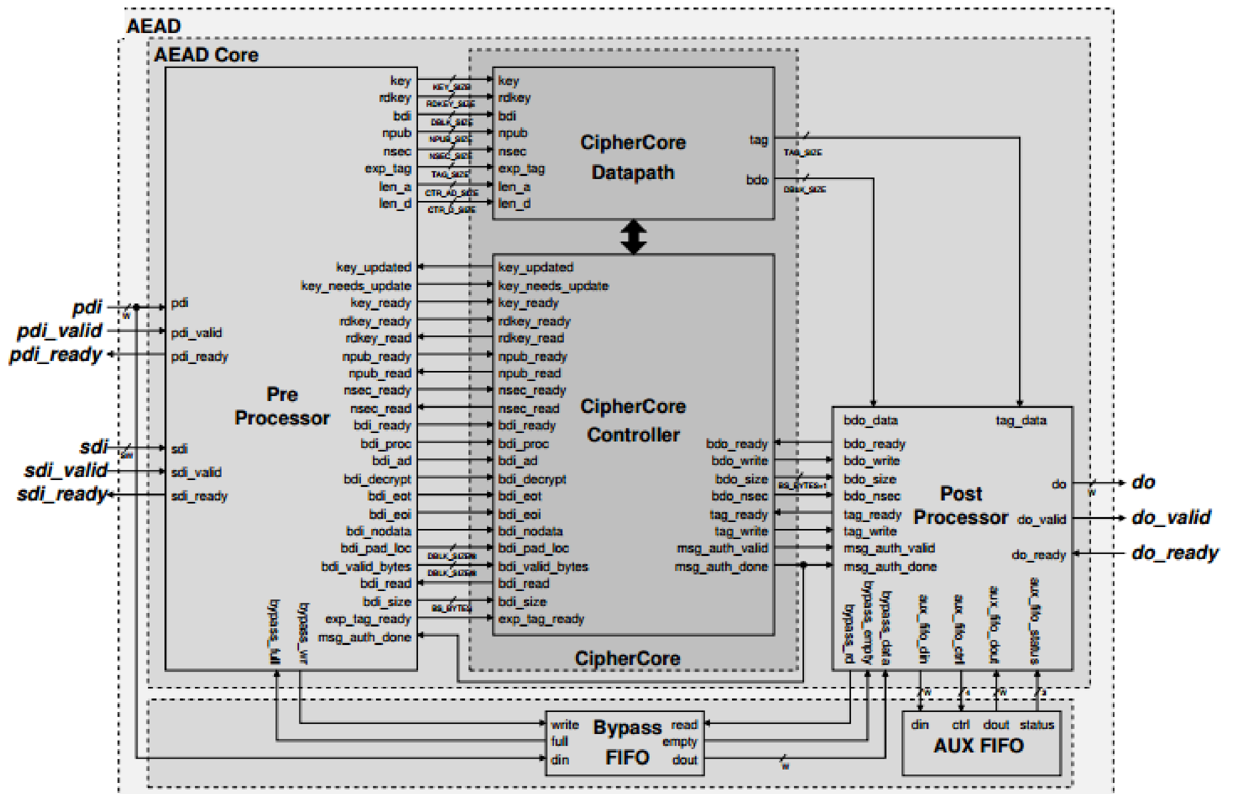


Figure 2-9: The AEAD interface [15]

In order to work with the AEAD interface, the ciphercore state machine is required to have the following state groups: *load/activate key*, *process AD, process message or ciphertext* and *generate or verify the authentication tag*. During the first state group*, load/activate key*, the state machine monitors the *key_needs_update* and *key_ready* inputs. *Key_ready* is high when the key is successfully loaded into the pre-processor. When the *Activate Key* instruction is received, the *key_needs_update* goes high. When both of these inputs are high, the ciphercore can read the key and set the *key_updated* output to high. This output remains high until a new key is needed.

After the key is read, the npub is next, using the *npub_ready* input to monitor its status and the *npub_read* output to move on to the processing the AD. Now the ciphercore needs to monitor the *bdi_ready*, *bdi_eot* and *bdi_eoi* inputs. For each block of AD read, the *bdi_read* output is set to high, and it waits for the next block until *bdi_ready* is high. This sequence repeats until the whole AD is read and the *bdi_eot* goes high. Then the message is read and monitored using the same inputs. During this group of states, the ciphertext is written to the post-processor using the *bdo_write* output. In the final group of states, the tag is generated and written to the post-processor using the *write_tag* command.

2.2.3 ATHENa & Design Compiler

ATHENa, the Automated Tool for Hardware EvaluatioN is aimed at a fair, comprehensive and fully automated evaluation of hardware cryptographic cores [16]. Results of comparison depend on the algorithms, but also on the selected hardware architectures, implementation techniques, the family of FPGA and used tools. ATHENa is an open-source tool that allows for an automated generation of optimized results for multiple hardware platforms. It is used alongside the AEAD hardware API in order to produce generalized, fair results. Each time ATHENa runs, it generates results for Xilinx and Virtex chip families. The results given are the area in slices and look up tables (LUT) and the timing in synthesis frequency and post place-and-route frequency. If selected, it can also present the best fitting device for said implementation.

Design Compiler by Synopsys is a logic-synthesis tool that is used in the design of an ASIC chip. It can be used to get the hardware area in µm, or in the amount of logic gates and timing in clock period. For this thesis, the free NangateOpenCell PDK 45nm library is used as the target and link library.

In this thesis, ATHENa is used to generate the FPGA results, whereas ASIC results are generated using Design Compiler. In Section 6, the Trivia-ck, Ketje and MORUS results are compared with other authenticated encryption algorithms from the ATHENa database.

3 Trivia-ck

3.1 Algorithm

Trivia-ck is an AE algorithm based on a stream cipher that uses a 128-bit key and a 128-bit nonce [17]. The nonce is divided into two 64-bit parts: param, which is a bitwise representation of the ck version, and the npub. The algorithm uses an internal state of 385 bits and outputs a 128-bit tag. Trivia-ck is described by the integrated combinations of the Trivia-SC and VPVHash modules. Trivia-SC is the stream cipher used to encrypt the message and VPVHash is used to generate the tag. The ck in the algorithm is used for the length of the intermediary tags, which get outputted after every 64-bit message block. The designers recommend two versions: Trivia-0 and Trivia-128. The first does not output the intermediary tags, while the latter outputs 128-bit intermediary tags [17]. A circuit diagram of Trivia-ck can be observed in Figure 3-1 below.

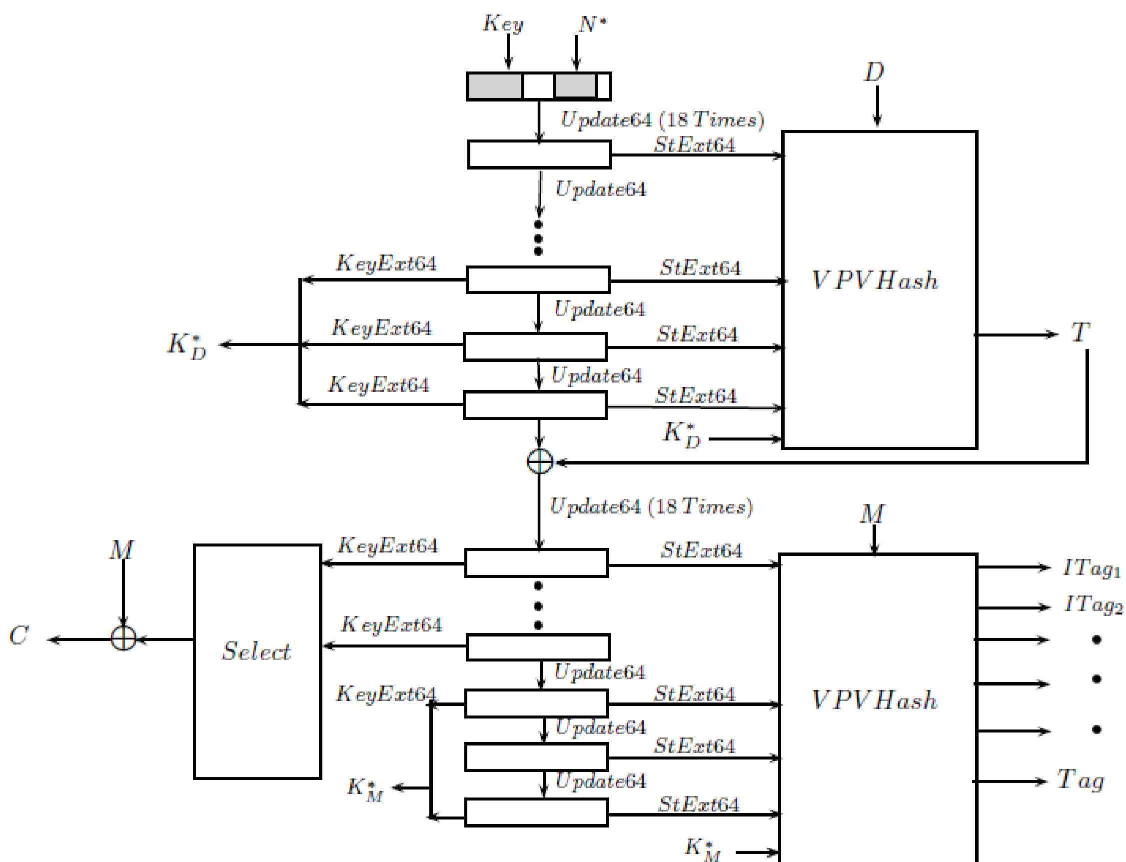


Figure 3-1: Trivia-ck circuit [17]

In the first step, both the key and nonce are loaded into the state. The state is divided into three non-linear feedback registers A, which is 132-bit, B, which is 105-bit and C, 147-bit. The key is padded with "1" and loaded into register A., while a padded IV gets loaded into C. B is initialised with "1".

After this, the state is updated for 1152 rounds. However, this updating can be parallelized up to 64 bits thanks to the flexibility of the Trivia-SC module, allowing the amount of rounds to be reduced to

18. Once the initialisation is done, the first 64 bits of the associated data is loaded into VPVHash alongside the first 64-bits of the stream (the StExt64 module in Figure 3-1). This step gets repeated until the whole associated data (AD) has been introduced. If the AD is not divisible by 64, the last block will be padded using 10* padding. If there is no AD, there will be a single 10^{63} "empty" block. After the AD step is done, the intermediary tag is generated in four extra clock cycles, by using the first and third state key and xoring this with the VPVHash output tag. This 160-bit intermediary tag is inserted into the state by xoring it with register A and the first 28 bits of B.

Before processing the message, the stream is updated again for 1152 rounds. When this is done, the message gets introduced in the same way as the AD (in 64-bit slices). The difference with the AD step is that with each 64-bit message block, a 64-bit ciphertext is outputted by xoring the message with the state key. After the message is fully introduced and the ciphertext is generated, the 128-bit tag is generated in three extra clock cycles. If ck is zero, then no extra tags are generated, when it is 128, the VPVHash tag is outputted every step [17].

The decryption works in a similar way, where the inputs are AD, nonce, Key, ciphertext and tag. By running the code with the ciphertext as the message, the outputted ciphertext will be the original message if the outputted tag is equal to the transmitted one.

It is important to note that this describes the first version of Trivia-ck. For the second CAESAR round, Trivia-ck has been updated to version two. In this version, the biggest change is that a 128-bit intermediary tag is generated instead of a 160-bit one, which allows the algorithm to use only one VPVHash block for both the intermediary tag and the final tag(s), saving more hardware space [18]. However the developer C-code for the second round have not been updated. A prototype has been designed for the second version in order to measure the differences against the first version, but these results are not necessary correct, since there is no way to verify the testbench results yet.

In Section 3.1.1 and 3.1.2, the Trivia-SC and VPVHash modules are examined more closely.

3.1.1 Trivia –SC

Trivia-SC is a variant of Trivium. This is a hardware oriented synchronous stream cipher that was designed for simplicity, without sacrificing security, speed or flexibility. It uses a 80-bit key and nonce and a 288-bit internal state and provides 80-bit security [19]. It is flexible in that the amount of iterations done in one clock cycle can be altered to suit the design. This is achieved by ensuring that any state bit is not used for at least 64 iterations after it has been modified. So it allows for up to 64 iterations to be calculated in parallel by duplicating the AND and XOR gates. This flexibility makes it easy to optimise for speed or area.

Table 3-1: Estimated gate counts of 1-bit to 64-bit hardware implementations [19].

Components	1-bit	8-bit	16-bit	32-bit	64-bit
Flip-flops	288	288	288	288	288
AND gates	3	24	48	96	192
XOR gates	11	88	176	352	704
NAND gates	3488	3712	3968	4480	5504

Trivia-SC extends Trivium by adding several extra modules, increasing the Key, IV and internal state size and introducing a non-linear effect in the key computation [17]. It extends the Trivium modules *Load*, *Update* and *KeyExt* by *StExt64* and *Insert*. *Load* inserts the 128-bit Key and IV into the internal state, *Update* updates the stream, KeyExt extracts the keystream from the state, *StExt64* extracts the first 64-bits from the state and finally *Insert* inserts an intermediary tag into the stream. The pseudocode of these modules can be found in the appendix. Just like Trivium, Trivia-SC is parallelizable up to 64 bits.

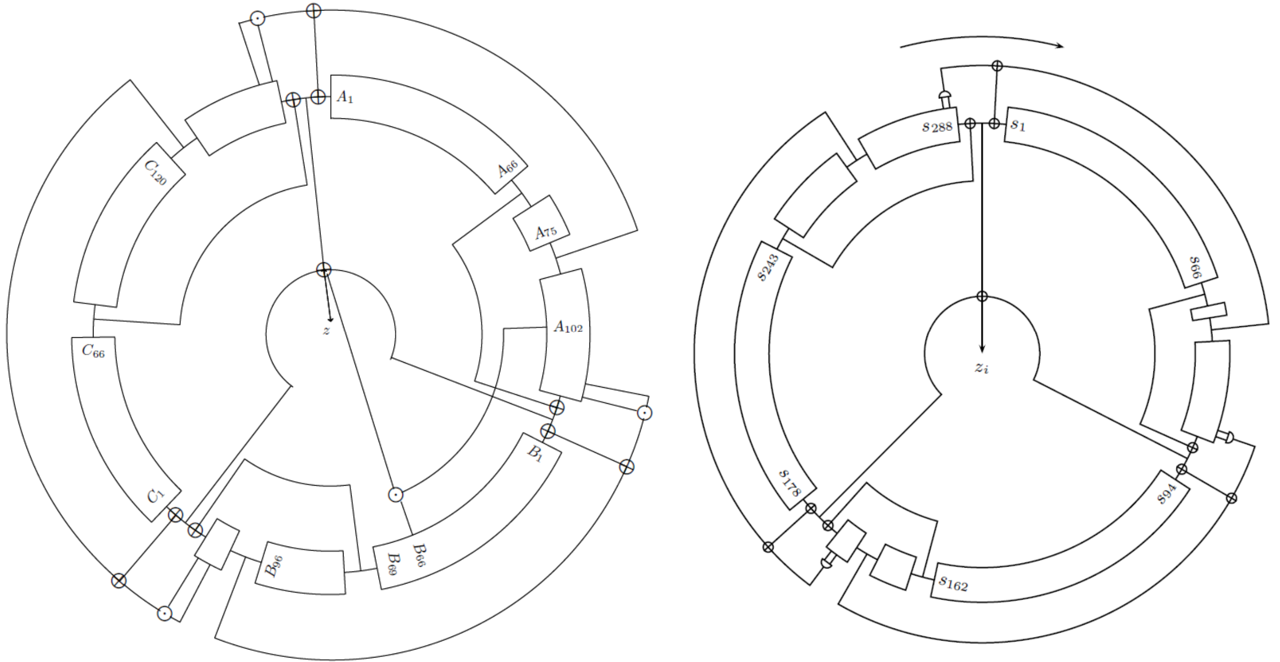


Figure 3-2: The Trivia-SC cipherstream (left) and the Trivium cipherstream (right) [17], [19]

3.1.2 VPV-Hash

The second component, VPV-Hash is used to compute the tag in three steps. The first step is calculating the checksum by applying Vandermonde based error-correcting code, called ECCode. Then it calculates the Pseudo-dot product of the checksum and the Trivia-SC state and finally, it calculates the Tag by using ECCode again [17].

ECCode is an error correcting code that extends its input by a distance d. The extra output is calculated via the Vandermonde Matrix and Horner's rule. The Vandermonde matrix is shown in Equation 3-1.

$$V_{n,d,l}\begin{bmatrix} 1 & \dots & 1 & 1 & 1 \\ \alpha^{l-1} & \dots & \alpha^{2} & \alpha & 1 \\ \dots & \dots & \dots & \dots & \dots \\ \alpha^{(l-1)(d-1)} & \dots & \alpha^{2(d-1)} & \alpha^{d-1} & 1 \end{bmatrix}$$

Equation 3-1: Vandermonde Matrix [17]

In this construction, the primitive polynomials used to represent the Galois fields GF(2^{32}) and GF(2^{64}) respectively can be found in Equation 3-2. The α used in Equation 3-1 is the primitive element of those Galois fields and is noted as α_{64} or α_{32}.

$$p_{32}(x) = x^{32} + x^{22} + x^2 + x + 1$$

$$p_{64}(x) = x^{64} + x^4 + x^3 + x + 1$$

Equation 3-2: Primitive polynomials of GF(2^{32}) and GF(2^{64}) [17]

The primitive element multiplication can be easily achieved using shift and bit-wise xor operations. The extended output y_1 to y_d is calculated through matrix multiplication of the Vandermonde Matrix with the original input, as shown in Equation 2 below.

$$\begin{pmatrix} y_1^k \\ y_2^k \\ \dots \\ y_d^k \end{pmatrix} = \begin{pmatrix} 1 & \dots & 1 & 1 & 1 \\ \alpha^{k-1} & \dots & \alpha^2 & \alpha & 1 \\ \alpha^{2(k-1)} & \dots & \alpha^4 & \alpha^2 & 1 \\ \dots & \dots & \dots & \dots & \dots \\ \alpha^{(d-1)(k-1)} & \dots & \alpha^{2(d-1)} & \alpha^{(d-1)} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_k \end{pmatrix}$$

Equation 3-3: Extended input generation [17]

In the first step of VPV hash, this extended output is the checksum and in the last step it is the tag. The pseudo-dot product is a multiplication of two 32-bit signals, one being the extracted key of the Trivia-SC block and the other the output of the first ECCode (which is either the associated data or the message extended with the checksum).

3.2 VHDL code

In this section, the design of the VHDL code is outlined in a "top-bottom" way. First the top level block is illustrated. Then the two subcomponents, the data path and the state machine are examined, as well as their underlying components.

3.2.1 Top level block

The code is divided into two parts, one part that takes care of the calculations, called the data path and another that controls the program flow, called the FSM. This is done to ease bug fixing and making adjustments, while also granting a better overview of the code. The top level block combines the inputs and outputs of these blocks together as shown in Figure 3-3. Since this top level block is similar for Ketje and MORUS, it will only be shown here.

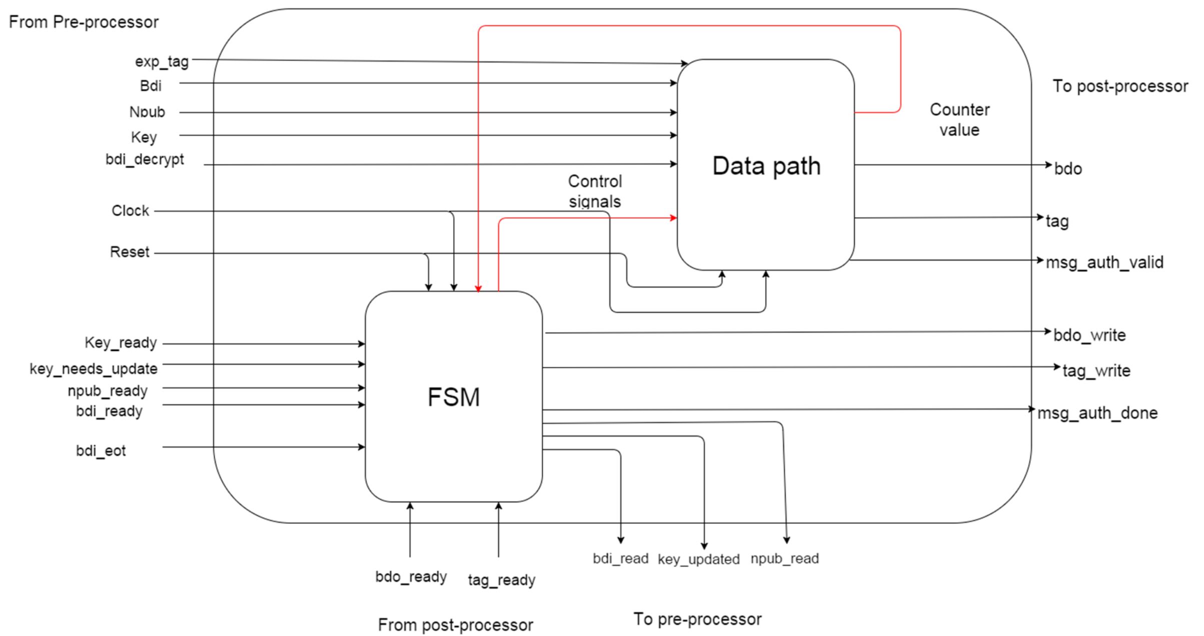


Figure 3-3: Trivia-ck ciphercore

3.2.2 Data Path

The data path contains all the functional units, such as logic gates and multipliers. It contains several smaller blocks, which represent the two mathematical components Trivia-SC and VPVHash. The data path is illustrated below in Figure 3-4.

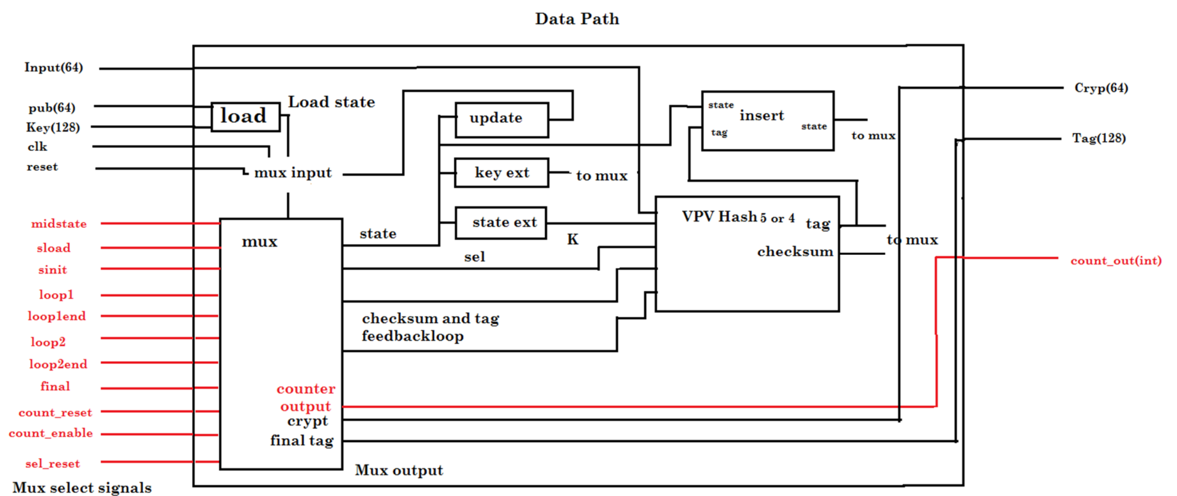


Figure 3-4: Trivia-ck data path

The load, update, key_ext, insert and st_ext blocks are all made up of simple registers, flip flops and logic gates. The multiplexer shown here is controlled by the state machine and routes the right signals to the right blocks and outputs. The *sload* signal routes the load block output to the state registers. *Sinit* and the *loop* inputs route the state registers to the update, key ext and state ext blocks. The *loop* inputs additionally routes the checksum and tag to the VpVhash block, while *loopend* routes the tag only.

The *midstate* inserts the tag into the state and *final* outputs the tag. During *loop2*, the ciphertext is also outputted. A counter is also used to count the rounds needed per state.
There are two version of the VPVHash block, one outputting a 160-bit tag and one outputting a 128-bit tag. They both contain several smaller components, shown in Figure 3-5.

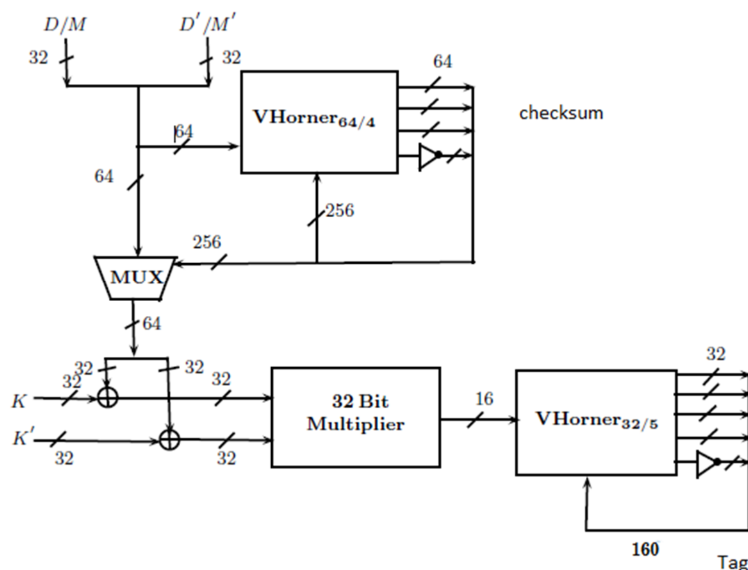


Figure 3-5: VPVhash [17]

As can be seen in Figure 3-5, VPVHash is built up out of two Vhorner blocks, a multiplexer and the 32-bit field multiplier. VHorner64 has two inputs: the 64-bit message or AD, and the 192-bit or 256-bit checksum, depending on the VPVHash block. The input checksum gets multiplied by the primitive element α_{64}, which is achieved by using a shift register. This new checksum then gets xored by the input to produce the output checksum. The output checksum then used as the new input checksum next clock cycle. The multiplexer first sends the message or AD in 64-bit chunks to the multiplier. After the last block, its 256-bit checksum is sent in 64-bit chunks. The 32-bit multiplier has two inputs, the 64-bit message or checksum from Vhorner64 gets xored with the 64 first bits of the cipher stream.

This result then gets split into two 32-bit parts to form the inputs. The multiplier is shown in Figure 3-6.

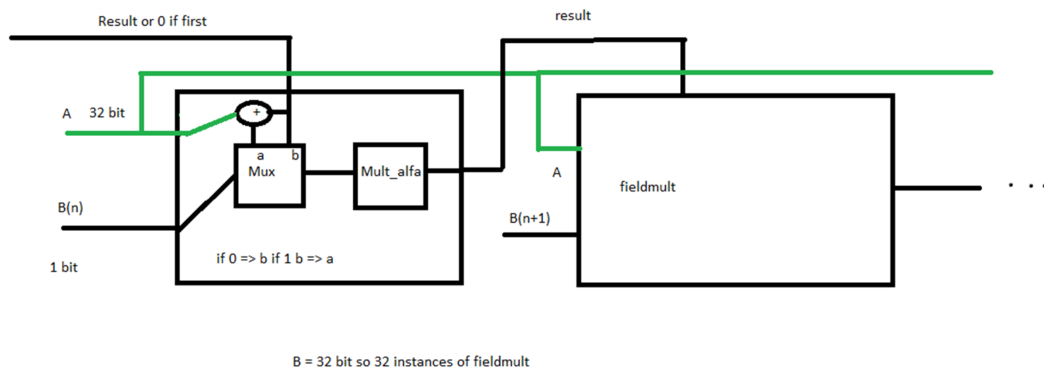


Figure 3-6: Field multiplier

The field multiplier includes a multiplexer and a primitive element shift register. As illustrated in Figure 3-6 above, there are 32 instances of this block, one for each bit in input B. If this bit is zero, the input result (which is zero for the first block and the output of the previous block for the other blocks) gets multiplied with α_{64}, otherwise the input is first xored with A and then multiplied with the primitive. This multiplication is done by shifting and xoring the bits in a specific way. The result is the output of the FM block. The final FM block is different from the other FM blocks in that there is no multiplication with α. The multiplier output is then used in Vhorner32, which works identical to Vhorner64 except it multiplies its input tag with α_{32} and outputs a 160-bit or 128-bit tag.

3.2.3 Finite State Machine

The state machine controls the large multiplexer in the data path, which in turn controls the internal signals. This state machine has 20 states, in which 16 states are used to cipher or decipher the message and four states which are needed to interact with the AEAD structure. The first two states wait for the AEAD wrapper to read the key and npub. After this, in the *load* state, the IV and key are loaded into the stream. The following state, *init*, initialises the stream. After this, during the *loop1* state, the AD is loaded in 64-bit chunks into the VPVhash. Each *loop1* state calculates the tag and checksum and routes it back into the Vhorner blocks next clock cycle. After all the AD are read, *loop1end* state is called. In this state the 160-bit tag is calculated. This takes four clock pulses because the mux output is 64 bits and the checksum is 256 bits. Also the streamkey of the first and last step are taken here, which are xored with the tag to calculate the intermediary tag. In the *midstate*, this intermediary tag is inserted into the stream. Then the process repeats itself for the message, where in every buffer state the 64-bit chunk of the ciphertext is written to the post-processor. In the *final* state, the tag is written to the post processor and the state waits for a new key. The state machine is illustrated in Figure 3-7.

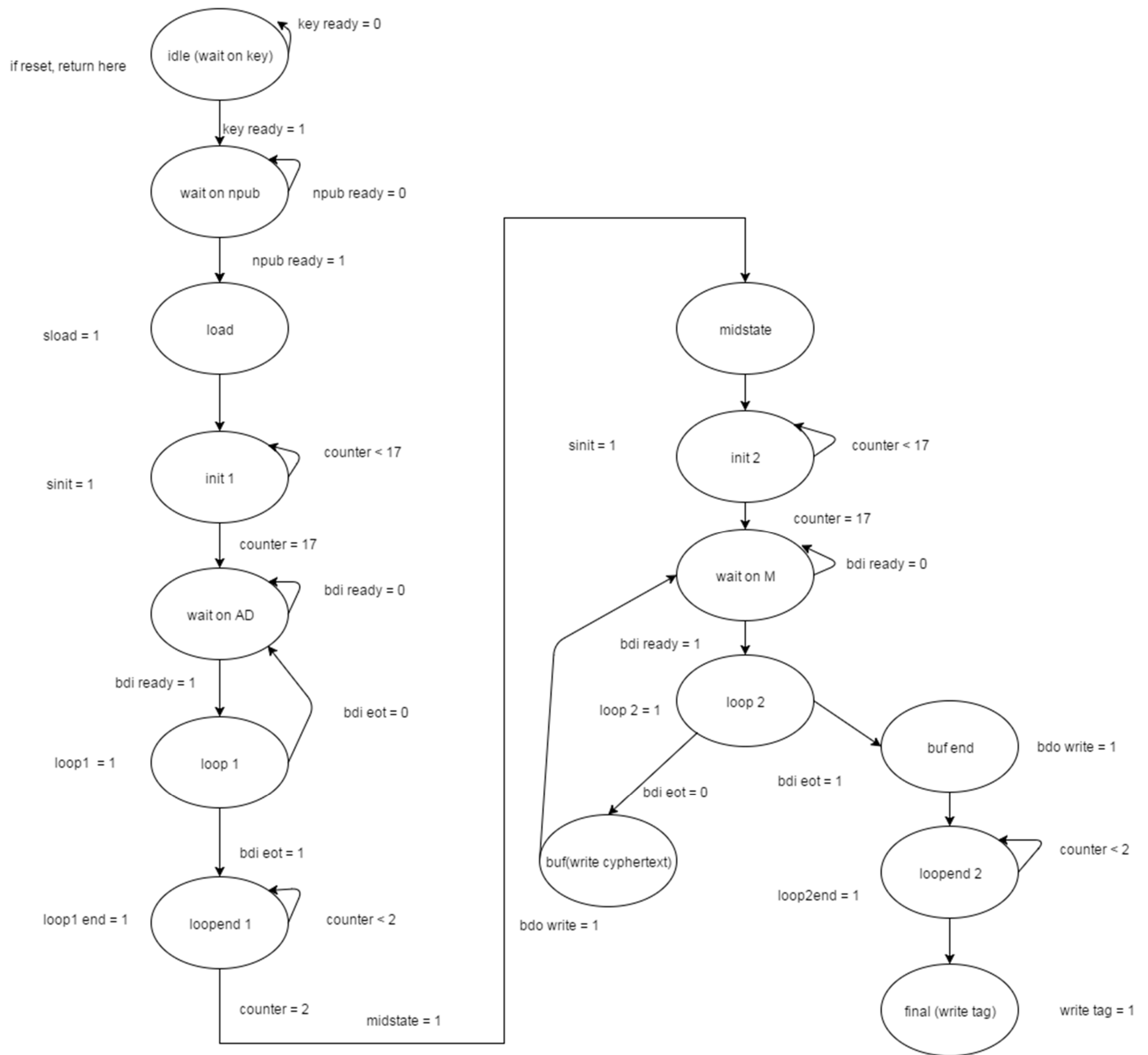


Figure 3-7: Trivia state machine

3.3 Trivia-ck optimisation

In order to test the algorithm fairly in both speed and hardware usage, the code has been improved using these two optimisation strategies. The first goal is to discover the areas which could use improvement. Here, both the multiplier in VPVhash and the parallelizable Trivia-SC module are obvious candidates. In the original code, 32 instances of the multiplier are used, resulting in a large amount of registers. It is also a critical path for the clock, since all calculations happen serial, decreasing the maximum frequency. Also, in the original code, the update and key exit block update 64 bits per clock cycle. This also requires a large number of registers. In the following paragraphs the original code, the area and speed optimisations are discussed. As mentioned before, the basic code and most optimisations were done on the first, outdated version of Trivia-ck. A prototype for the new version of the code is discussed in Section 3.3.4. The results for area usage and timing can be found in Section 3.3.5.

3.3.1 Basic code

In order to observe the impact of the optimisations on the algorithm, the basic version is discussed first. In this version, no optimisation path is followed and the blocks use their recommended parameters. This version uses 64-bit parallel update and key generation, 32 instances of the multiplier without pipelining and two VPVhash. It also uses a normal binary counter and 5:1 multiplexers in the VPVhash block. Shown below are the testbench results of the ciphercore for a 15 byte AD and a 16 byte message, resulting in three 64-bit ciphertext chunks. The results for this version are found in Section 3.3.5.

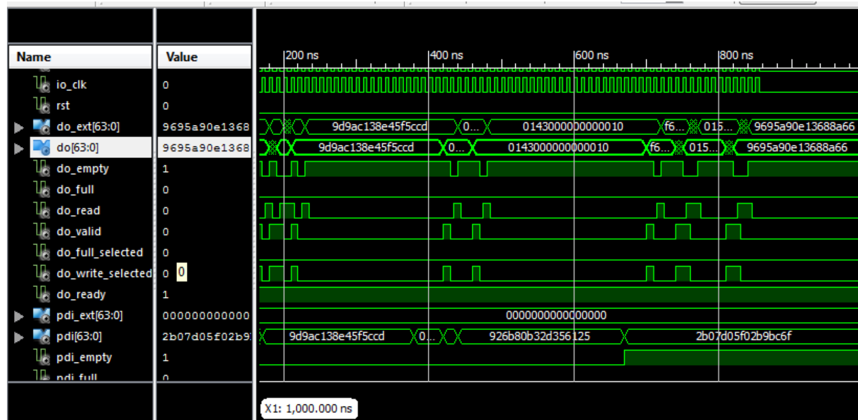


Figure 3-8: Testbench results from the basic Trivia-ck version

3.3.2 Area optimisations

This section outlines all the area reducing optimisations. The performed optimisations are: rolling up on the 32-bit multiplier, reducing 64-bit update to 1-bit and changing the multiplexer. The Vhorner block was also rolled up, but this resulted in more area, so this change was ultimately undone.

The first optimisation is rolling up the 32-bit multiplier. The original field multiplier block is built up out of 31 identical FM blocks and one final FM block. All of these blocks are connected using 32 32-bit signals.

The rolled up version of the field multiplier only uses one FM block and the FM last block, which are connected using only three 32-bit signals. The FM blocks in the original code differ only in the bit of B they monitor. By using a counter, the 32 bits of B are sent to the FM block bit by bit. The *result_out* output is routed back to the input every clock cycle. After 31 cycles, the FM last block output is the final result. For every time the multiplication result is needed, the code needs 32 clock cycles to process it, which is during every *loop* (processing AD and message) and *loopend* (tag generation) state. This results in higher delay and latency, which can be calculated using Equation 3-5:

$$encryption\ delay\ (cycles) = 262 + \frac{Mlen}{64} \cdot (1 + 32) + \frac{ADlen}{64} \cdot (1 + 32)$$

In order to further reduce the registers, VHorner was also rolled up. In the basic code, each VHorner uses four internal registers of 128-bit, 160-bit 192-bit or 256-bit. From each of these registers, 32 or 64 bits are taken to form the tag or checksum, after being multiplied by α. In order to reduce the code to use only one register, VHorner was rolled up, so that the multiplication and checksum or tag would be generated in four cycles. However, the added control scheme needed for this increased the total area, so this optimisation was not used.

The second optimisation is reducing the amount of XOR- and AND-gates used in the update and key_ext blocks. This is achieved by only updating one bit at a time instead of 64-bits parallel, which is possible since Trivia-SC ensures that any state bit is not used for at least 64 iterations after it has been modified. Now 64 clock cycles are needed every time the state is updated. This means 2304 clock cycles are needed for the two initialisation states, and 64 extra for every message and AD block and 448 extra for the tag generation.

The last area focussed optimisation is changing the multiplexers used in the VPVHash blocks. In the original code, the two VPVHash blocks use a 5:1 multiplexer with a 3-bit select signal. During synthesis, such a multiplexer is built up out of four 2:1 multiplexers (N-1 2:1 mux, where N is the amount of inputs). Here, each 2:1 mux requires a separate LUT, which results in four LUTs [20]. It is however more efficient to use one 4:1 mux and one 2:1 mux to create the 5:1 mux. The 4:1 mux used two LUTs, which results in three LUTs total. A 4:1 is built as depicted in Figure 3-9 below.

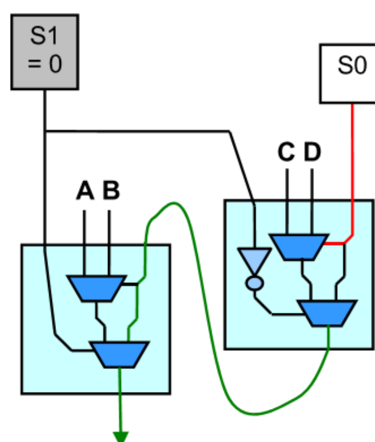


Figure 3-9: A 4:1 mux using 2 LUT's [20]

In the VPVHash4 block, the multiplexer only has four inputs, which allowed a single 4:1 mux after changing the code. This change resulted in better area but also better timing, so these mux trees are also used in the speed optimised version.

A lot of the optimisation increased the total amount of clock cycles needed for encryption and decryption, which results in more states in the state machine. The new state machine is shown in Figure 3-10.

The loading of the key and npub remains the same. The initialisation state is used to update the stream. After loading the key and npub, the stream now gets updated 1152 times, since it now happens bit by bit.

After this, the FSM waits on the AD blocks. After each block, the 32-bit multiplication is calculated over 32 clock cycles during the *loadvpv* state. After this, the Pre-processor is notified that the block is

read and the stream is updated 64 times in the init state. When all the AD blocks are read, the init state continues to the *incsel* state. This state increments the select signal for the multiplexers in the VPVHash blocks. Thereafter, the multiplication is calculated in the *loadvpvend* state and the tag is calculated in the *loopend2* state. Then, the state is updated again for 64 times. This gets repeated four times to calculate the valid tag. The processing of the message blocks is similar. The *update_C* state generates the ciphertext and is called after updating the stream, after the message block has been read.

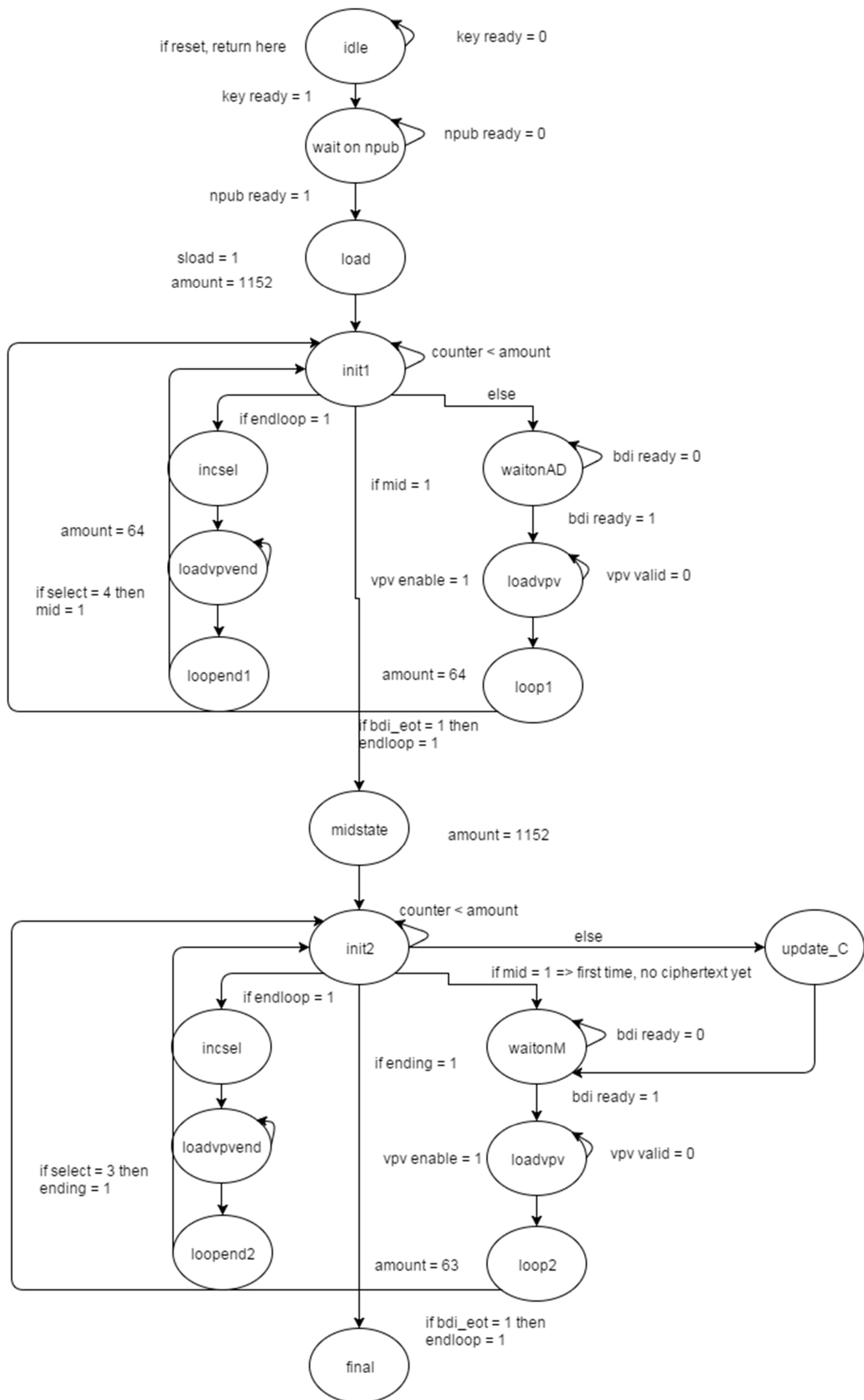


Figure 3-10: state machine for the area optimisation

3.3.3 Speed optimisations

This section outlines all the speed optimisations. The performed optimisations are: pipelining the field multiplier, changing the VPVHash multiplexers and selecting a better counter. Also, several registers where duplicated in order to shorten the path delays.

The first optimisation is reducing the critical path by pipelining the field multiplier. This path starts in the data path control multiplexer input. Whenever *loop1, loop2, loop1end* or *loop2end* are high, the tag gets generated in the VPVhash block. Here, the field multiplication takes place. In the basic version, 32 of these FM blocks are utilised every clock pulse. Since each of these blocks uses a 32-bit 2:1 multiplexer, using them all at once slows the code down considerably. However, this critical path can be broken up using intermediary registers. This technique is called pipelining. Although a pipelined circuit requires more clock cycles to complete, the maximum clock frequency increases, which results in higher throughput. The benefits of pipelining can be explained using a simple example: if 5 bits of data have to be transferred over a path with a 2 ns delay, bit for bit it will take 10 ns to complete. When this path is pipelined, the path delay is lowered, for instance to 1 ns. It will however take one extra clock cycle to get the final bit, thus 6 cycles are needed. So the data only takes 6 ns as opposed to the 10 ns it took in the non-pipelined path. In this optimisation, 31 extra registers are implemented, one between every FM block.

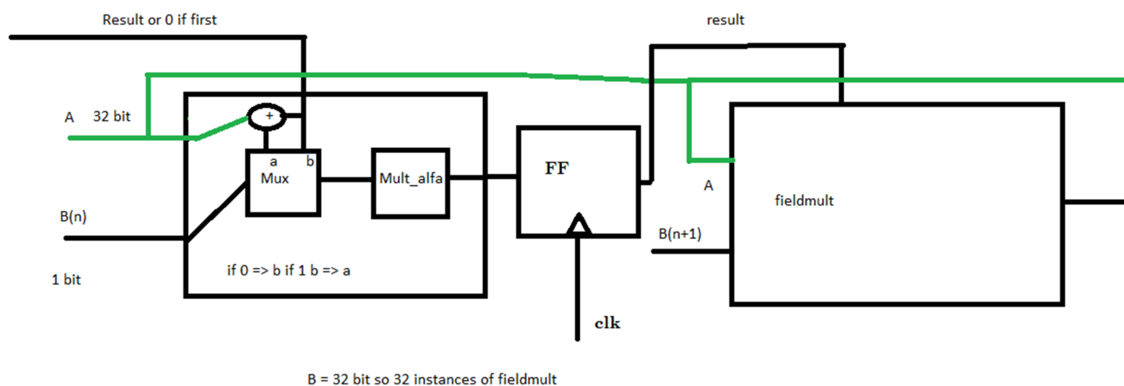


Figure 3-11: Pipelined field multiplier

It takes 32 clock pulses to get the multiplier output to the VHorner4 block to generate the tag. However, since the tag is only needed during the *midstate* and the *final* state, the total added delay is only 64 clock pulses, independent of the AD and message length.

Another change is switching from the 5:1 multiplexer to the 4:1 and 2:1 multiplexer, as explained in the area optimisations. This optimisation also results in faster clock frequencies.

The counter used in the state machine is also changed from binary to one-hot. In the basic code, this timer is written as shown below.

```
counter: process(clk,count_reset,count_enable)
 begin
   if (count_reset = '1') then
             count_out_buffer <=(others =>'0');
   elsif (rising_edge(clk)) then
              if(count_enable = '1') then
                        count_out_buffer <= count_out_buffer+1;
             end if;
   end if;
end process;
```

And this is how the counter is read:

```
If (count_in < 3 ) then
```

Here, the code needs to check several bits in order to compare the count in value with the number. This takes extra time [21] . The process can be increased in speed if the code only needs to compare a single bit instead like so:

```
if (count_in(3) = '1') then
```

In order to achieve this, the counter is changed form a normal incrementing counter to a shift register, where a '1' is loaded as the LSB (last significant bit) and this '1' gets shifted to the left every clock pulse, like demonstrated below.

```
counter: process(clk,count_reset,count_enable)
 begin
   if (count_reset = '1') then
             count_out_buffer(35 downto 1) <=(others =>'0');
             count_out_buffer(0) <= '1';
   elsif (rising_edge(clk)) then
              if(count_enable = '1') then
             count_out_buffer <= count_out_buffer(34 downto 0) & '0';
             end if;
end if;
end process;
```

In order to give the pipelined field multiplier time to compute the product, the state machine has been altered. Two extra states, *update T* and *update Tag*, are added between the *loopend1* and *midstate*, and between the *loopend2* and *final* state. These states are 32 cycle loops that flush the pipelined field multiplier, so that the right tag is outputted in the next state. The altered state machine can be seen in Figure 3-12.

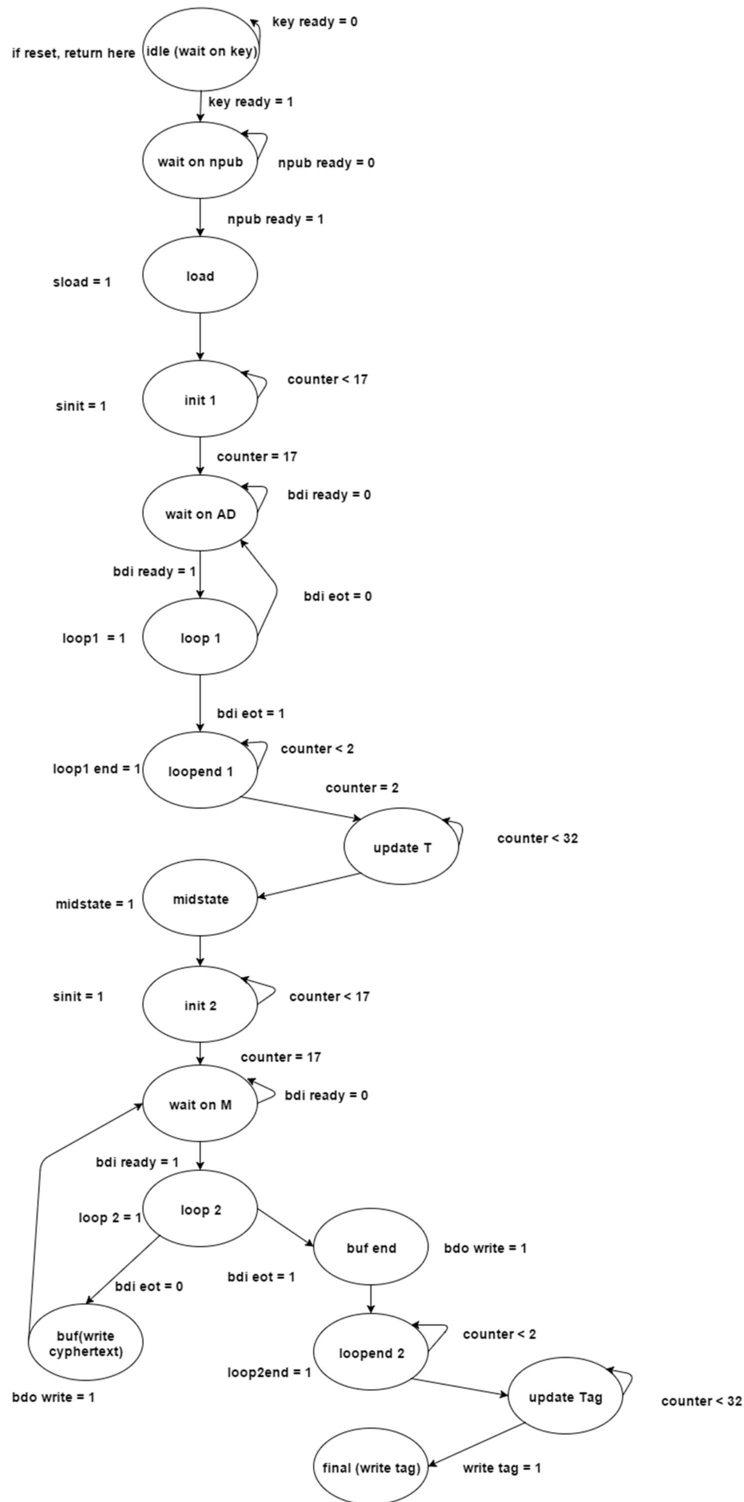


Figure 3-12: state machine of the speed optimised code

3.3.4 Trivia-ck version 2

In this section, a prototype of the second round Trivia-ck is discussed. As mentioned, this prototype is untested since there is no developer C-code yet. The second version differs from the first in several ways. The first change is in the way the key an npub are loaded into the state. The C part of the state is now loaded with zero's and three "1", instead of all "1". The important change is however the reduction of the intermediary tag to 128-bit instead of 160-bit. This change also removes the use of two different VPVhash blocks, now only VPVhash4 is needed for both tag generation steps. This change decreases the amount of registers considerably. The third change is the way the A and B inputs to the multiplier are formed. Now the key and multiplexer output are divided into four equal parts, and the first and third parts from both are xored to form A, and the second and fourth parts are xored to form B. The final change is that the second keyext output during tag generation (loop1end and loop2end) is multiplied with α_{32} [18].

Only the reduction of the intermediary tag changes the code in a drastic way. As mentioned, now only one VPVhash block is needed. This also simplifies the insert block, and removes the need to save the keyext outputs, since now the same outputs are used in both tag generation phases. The Trivia-ck prototype is built using the Trivia-ck speed optimised version, so the state machines are the same.

3.3.5 Results

Here, the results on area and speed are compared between the different versions of Trivia-ck. Shown below are the result of the area usage, calculated using the ATHENa tool for the basic Trivia-ck code. LUTs are look-up tables and FF's are flip flips. The first results with the full AEAD interface are tested on Spartan and Virtex .

Table 3-2: Area results for basic Trivia-ck using AEAD wrapper

Device	LUTS	SLICES	FFs
Spartan 3	5482	3116	2192
Spartan 6	7435	2223	2321
Virtex 5	7491	2398	2310
Virtex 6	7289	2093	2295

However, the results above include the extra hardware of the pre- and postprocessors. In order to truly judge the algorithm for its area usage, it has also been tested without the AEAD wrapper. For this purpose, the Ciphercore wrapper provided by the ATHENa website has been used.

Table 3-3: Area usage for basic Trivia-ck without AEAD wrapper

Device	LUTS	SLICES	FFs
Spartan 3	5071	2959	2141
Spartan 6	3540	1120	2002
Virtex 5	3407	1075	1970
Virtex 6	3322	1399	1965

As can be seen, for Spartan 3, the wrapper only uses about 5,04% of the total usage. However in Spartan 6, the usage is 49;61%, for Virtex 5 55,17% and Virtex 6 33,16%. In future results, only the area usage and speed of the algorithm using the Ciphercore wrapper is calculated. Shown below are the results on area usage for both the area optimised and speed optimised versions and Trivia-ck v2.

Table 3-4: Area results for area optimised Trivia-ck

Device	LUTS	SLICES	FFs
Spartan 3	2310	1593	2282
Spartan 6	1775	618	2113
Virtex 5	1974	711	2132
Virtex 6	1724	699	2112

Table 3-5: Area results for speed optimised Trivia-ck

Device	LUTS	SLICES	FFs
Spartan 3	4823	4302	7216
Spartan 6	4375	1405	6198
Virtex 5	4254	1665	6166
Virtex 6	3981	1205	6153

Table 3-6: Area results for Trivia-ck v2

Device	LUTS	SLICES	FFs
Spartan 3	3020	2525	3980
Spartan 6	2572	777	3365
Virtex 5	2760	1086	3346
Virtex 6	2312	808	3369

It can be clearly seen that the area optimised version has the lowest area usage. The speed optimised version has more slices and FF's than the basic version, because of the extra added pipeline registers. The Trivia-ck v2 version sees a significant reduction on area when compared to the speed optimised version, on which it is based. This reduction can be credited to the removal of one of the VPVHash blocks.

Tables 3-7 to 3-10 show the timing results for the basic version, area optimised version, speed optimised version and the second version of Trivia-ck. The place and route (PAR) frequency is the highest clock frequency at which the chip can run the algorithm.

Table 3-7: Timing results for basic Trivia-ck

Device	PAR Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
Spartan 3	48,57	1554,24	988,3+41,1*(M+AD)
Spartan 6	94,67	3029,44	507+21,1*(M+AD)
Virtex 5	120,57	3860,07	397,9+16,6*(M+AD)
Virtex 6	126,41	4045	379,7+15,8*(M+AD)

Table 3-8: Timing results for area optimised Trivia-ck

Device	PAR Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
Spartan 3	120,42	79,45	20993+805,5*(M + AD)
Spartan 6	149,61	98,71	16897+648,4*(M + AD)
Virtex 5	196,97	129,96	12834+492,5*(M + AD)
Virtex 6	237,70	156,83	10635+408,1*(M + AD)

Table 3-9: Timing results for speed optimised Trivia-ck

Device	PAR Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
Spartan 3	121,12	3875,84	899,9+16,5*(M+AD)
Spartan 6	146,59	4690,88	743,6+13,6*(M+AD)
Virtex 5	177,49	5679,68	614,1+11,3*(M+AD)
Virtex 6	235,07	7522,24	463,7+8,5*(M+AD)

Table 3-10: Timing results for Trivia-ck v2

Device	PAR Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
Spartan 3	120,15	3846,15	907+17*(M+AD)
Spartan 6	126,89	4060,40	859+16*(M+AD)
Virtex 5	150,24	4807,69	726+13*(M+AD)
Virtex 6	224,67	7189,40	485+9*(M+AD)

In order to measure the speed of the algorithm, the throughput and the latency are calculated. The throughput of the cipher is the amount of bits per second that are encrypted. The latency is the amount of time it takes for the full message to be encrypted or decrypted. Equation 3-4 shows the method used to obtain the throughput.

$$Throughput\left(\frac{Mbit}{s}\right) = \frac{Blocksize}{latency(N+1, Tclk) - latency(N, Tclk)}$$

Equation 3-4: Throughput calculation [22]

Trivia has a constant block size of 64-bit for all optimisations. The formula in the denominator of Equation 3-4 is the processing time of 1 message block. First the total processing time for N+1 blocks is calculated and the processing time for N blocks is subtracted. This results in the processing time of one block. This processing time needs to be calculated using the amount of clock cycles that are needed to process one block.

Equation 3-4 is used to calculate the throughput for long messages. For short messages, the effective throughput is smaller. The formula for throughput for short messages is found in Equation 3-5. Since the equation for latency is given in Equation 3-7, and for each of the selected candidates the latency per message and AD block is calculated, sufficient information is provided to calculate and compare the throughput of different message sizes. In this thesis, only Equation 3-4 is used to calculate the throughput.

$$Throughput\left(\frac{Mbit}{s}\right)_{Eff} = \frac{N \cdot Blocksize}{latency(N, Tclk)}$$

Equation 3-5: Throughput calculation for short messages [22]

The equation used in calculating the delays can be found below.

$$Delay\ (cycles) = initialisation + tag\ generation + M\ processing \cdot \frac{Mlen}{blocksize} + AD\ processing \cdot \frac{Adlen}{blocksize}$$

Equation 3-6 : Delay calculation in cycles

The amount of cycles it takes to process an input depends on several factors. One factor is the AEAD wrapper delay. The code needs to wait until the input, key and IV get updated. The amount of time this takes depends on the length of the inputs and size of the chip IO- buffers. When taking this delay as the absolute minimum, presuming only one clock pulse for each AEAD state, it can be calculated using Equation 3-5:

$$AEAD\ delay\ (cycles) = 3 + \frac{Mlen}{64} + \frac{Adlen}{64}$$

Three states: *idle*, *wait_on_npub* and *final* are used to input the key, npub and write the tag and need at least 1 clock pulse. *WaitonAD* and *WaitonM* are repeated for every 64 block of input.

The encryption delay is the actual amount of cycles needed process a full message without the AEAD delay. This delay depends on the structure of the state machine and on the message/AD length. For the basic version, it is calculated as following:

$$encryption\ delay\ (cycles) = 45 + \frac{Mlen}{64} + \frac{ADlen}{64}\ (3)$$

The 45 constant clock cycles are: the loading of the key and IV into the cipher stream (one cycle), two initialisation states (18 cycles each), the *midstate* (one cycle) and two tag generation states (which take three and four cycles respectively). For each 64-bit message/AD block one cycle is needed to process the data.

Thus the total delay is:

$$total\ delay\ (cycles) = AEAD\ delay + encrytion\ delay = 48 + 2\frac{Mlen}{64} + 2\frac{ADlen}{64}$$

Using the maximum frequency, the amount of cycles/s can be calculated, and thus the delay in ns.

$$latency\ (ns) = total\ delay\ (cycles) \cdot Tclk$$

Equation 3-7: Latency

Now the throughput can be calculated by calculating the latency using Equation 3-7 for N+1 blocks and N blocks, and using the result in Equation 3-4.

In the area optimised version, the encryption delay is calculated as such:

$$encryption\ delay(cycles) = 2528 + \frac{Mlen}{64} \cdot (1 + 32 + 64) + \frac{ADlen}{64} \cdot (1 + 32 + 64)$$

And in the speed optimised and second version, it is calculated as following:

$$encryption\ delay(cycles) = 109 + 2\frac{Mlen}{64} + 2\frac{ADlen}{64}$$

From the results, it can be observed that the speed optimised version has the highest throughput. The PAR frequencies have increased, compared to the basic version, thanks to the pipeline shortening the critical path. Because of this, the throughput has also increased, because the message block processing time has remained the same. Trivia-ck version 2 has a lower throughput because of the more complex multiplier inputs. These results are however untested and are thus susceptible to change. The area optimised version has the highest frequency, being smaller in size allows the components to be placed closer together, shortening the path delays. However, since the latency is high, the actual throughput is low.

Lastly, shown below are the ASIC results using the NanGate PDK 45nm library for all Trivia-ck versions. The area is given in both micron square and gate equivalent (GE), showing the estimated number of gates needed.

Table 3-11: ASIC results for basic Trivia-ck code

Area (µm^2)	Area (kGE)	Timing crit path (ns)	Max Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
21049,91	26,31	4,4264	225.92	7209,28	213+9*(M+ AD)

Table 3-12: ASIC results for Trivia_ck area optimisation

Area (µm^2)	Area (kGE)	Timing crit path (ns)	Max Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
16082,89	20,1	2,5244	396.13	261,37	6382+245*(M+ AD)

Table 3-13: ASIC results for Trivia_ck speed optimisation

Area (µm^2)	Area (kGE)	Timing crit path (ns)	Max Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
63868,20	80	2,7728	360.64	11540,7	302 + 5,5*(M + AD)

Table 3-14: ASIC results for Trivia_ck v2

Area (µm^2)	Area (kGE)	Timing crit path (ns)	Max Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
33625,86	26,31	2,3667	422,52	13520,64	258+8*(M+ AD)

4 Ketje

4.1 Algorithm

Ketje is an AE function that is aimed at memory-constrained devices. It relies strongly on nonce uniqueness for security [23]. There are two suggested versions of Ketje: KetjeJr and KetjeSr. Their key, nonce and state sizes can be found in Table 4-1.

Table 4-1: Ketje parameters

Recommended version	Key (bits)	Nonce (bits)	State (bits)	Block length (bits)
KetjeJr	96	80	200	16
KetjeSr	128	128	400	32

KetjeSr and KetjeJr are built on round-reduced versions of KECCAK-f[400] and KECCAK-f[200], which are called by a construction named MonkeyDuplex. This variant of a duplex construction supports different types of calls that invoke it with different numbers of rounds. On top of MonkeyDuplex is the MonkeyWrap construct. Section 4.1.1 will first outline the Sponge construction to get a better understanding of the cipher. Sections 4.1.2 to 4.1.4 will explain the MonkeyWrap, MonkeyDuplex and KECCAK-p constructions.

4.1.1 Sponge construction

This sponge construction is a mode of operation that takes a binary string of any length as input and returns a binary string of any required length. The construction has a fixed length permutation or transformation *f* and a finite internal state with a number of bits equal to b (the width of the state). B is divided into two sections, one with size r (called the bitrate) and the other with size c (called the capacity).

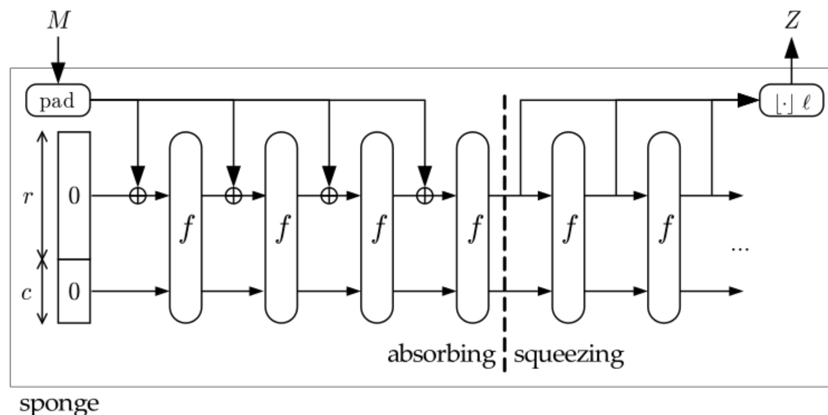


Figure 4-1: The sponge construction [14]

First the input string gets padded with a reversible padding rule. Then it gets cut into equal length blocks of length r. The state is initialised to zero and the construction continues in two distinct phases:

the absorbing phase and the squeezing phase. In the first phase, the r-length input blocks get xored into the state. Following each block, the function *f* is called. After all the input blocks are absorbed, the squeezing phase begins. Now the first r bits of the state are the output blocks. The amount of output blocks can be chosen by the user. The last c bits of the state are never affected by the input and never outputted during the squeezing phase. It can be observed in Figure 4-1 that first the full input is absorbed before the output appears. A variant of the Sponge, the Duplex, alternates between absorbing and squeezing for every input block. It is this construction that is used in Ketje and other single pass AE systems [14].

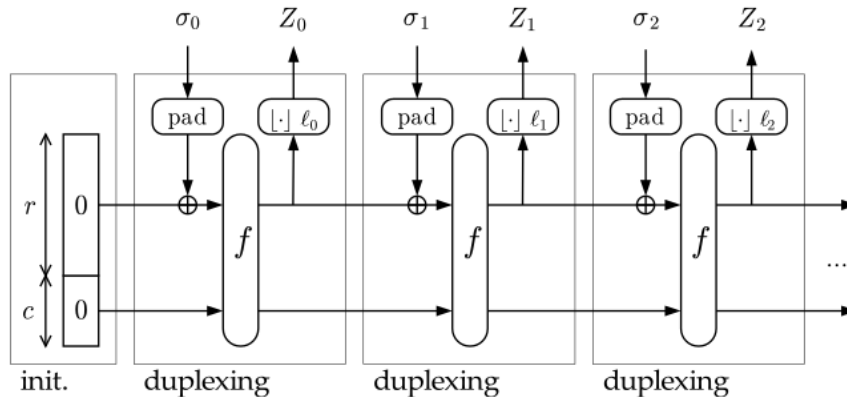


Figure 4-2 : Duplex construction [14]

4.1.2 MonkeyWrap construction

MonkeyWrap is the top level construction of Ketje and takes care of wrapping and unwrapping the data. When wrapping, it takes the AD and message and returns the ciphertext and tag. The opposite is called unwrapping, which takes ciphertext, AD and tag and returns the message is the tags match.

The authenticated encryption process starts with initializing the state by loading the key and nonce. The key first gets packed in a keypack. Here the key gets extended by 16 bits. The first eight bits represent the length of the keypack, which is the key size plus 16, in bytes. The last eight bits of the pack are simple 10* padding. This keypack then gets appended with the nonce, after which the MonkeyDuplex gets called with 16 rounds.

After the initialisation is done, the AD gets absorbed in the state in blocks of 16-bit for KetjeJr and 32-bit for KetjeSr. Each AD block gets padded with two frame bits "00" before calling MonkeyDuplex with one round. This process repeats until the final AD block. This block gets padded with "01" instead, and the first 16-bit or 32-bit (length r) of the state gets xored with the first message block to obtain the first ciphertext block.

Following this, the message gets absorbed into the state in either 16-bit or 32-bit blocks after being padded with the frame bits "11". Each MonkeyDuplex output gets xored with the next message block to form the ciphertext.

The final message block is used to generate the tag. First it gets padded with "10" and the MonkeyDuplex gets called with six rounds. The 16 or 32 first bits from the state form the first bits of the tag. The rest of the tag gets generated by inputting a "0" bit into the stream and duplexing one round. Each time 16 or 32 bits are taken from the state until the tag is 90 or 128 bits [23].

Unwrapping or decrypting works identical. The AD, ciphertext and tag return the message if the input tag equals the output tag. An illustration of the MonkeyWrap construction can be seen in Figure 4-3.

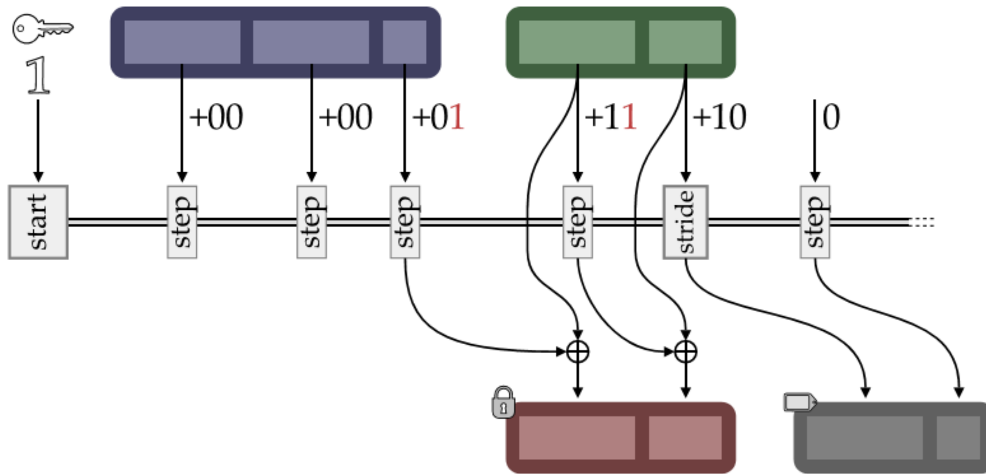


Figure 4-3: Encrypting data using MonkeyWrap. The blue blocks represent the AD, green the message, red the ciphertext and grey the tag [23]

4.1.3 MonkeyDuplex construction

As mentioned before, MonkeyDuplex is a variant of a Duplex construction in that the amount of rounds can be selected before the r-bits of the state are outputted. MonkeyDuplex has three modes: start, step and stride.

In start mode, which is during the initialisation phase of MonkeyWrap, the keypack appended with the nonce is the input. This input gets padded to match the state width b (200-bit for KetjeJr and 400-bit for KetjeSr), using 10*1 padding (so padding two extra bits would be "11", three would be "101"etc.). This padded input gets xored into the state, after which the KECCAK-p permutation gets called with 12 rounds to initialise the state.

In step mode, called during the AD and message wrapping/unwrapping phase, the message or AD gets padded further to match the length of r using 10*1 padding. Ketje's bitrate length is equal to the block size plus four, so for KetjeJr this is 20 bits and for KetjeSr this is 36 bits. This padded input gets xored into the state. Now the KECCAK-p permutation gets called with one round, and the r-bits of the state are used to calculate the ciphertext.

Stride mode is identical to step mode, the difference being that the KECCAK-p permutation gets called with six rounds before the state output is used. Figure 4-4 illustrates the MonkeyDuplex construction.

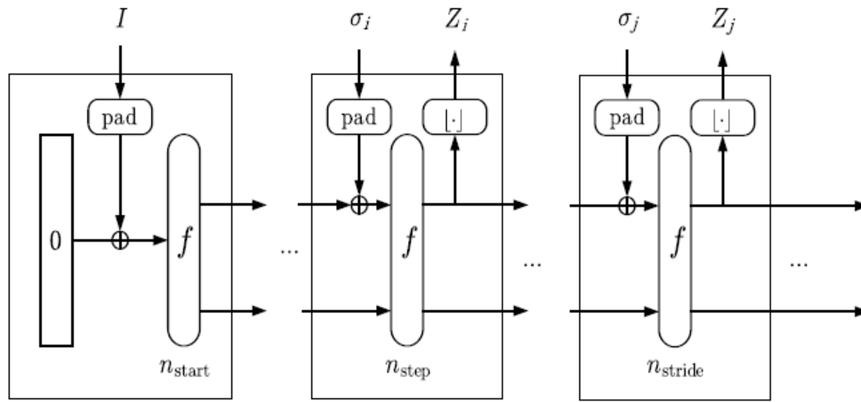


Figure 4-4: The MonkeyDuplex construction [23]

4.1.4 KECCAK-p permutation

KECCAK-p is a fixed length permutation defined by its width b and number of rounds n_r. It is derived from the KECCAK-f permutation, in that KECCAK-p is the application of the last n_r rounds of KECCAK-f [23]. KECCAK-p is a sequence of operations on a state S, which is a three-dimensional array of binary numbers. S is described as s[5,5,w], where w is the width b divided by 25. In the case of KetjeJr, w is eight and for KetjeSr this is 16. The input bit string is mapped to the state using Equation 4-1.

$$input[w \cdot (5y + x) + z] = s[x, y, z]$$

Equation 4-1: Input string to 3d matrix mapping [23]

Expressions taken at the x and y coordinates are taken at modulo five and those at the z coordinates are taken at modulo w.

Each KECCAK round consists of five steps: theta, rho, pi, chi and iota. The pseudocode for each of the steps can be found in the appendix.

Sections 4.1.4.1 to 4.1.4.5 describe each of the steps. Before continuing, it is important to know the naming conventions for the KECCAK-p state, which are depicted in Figure 4-5.

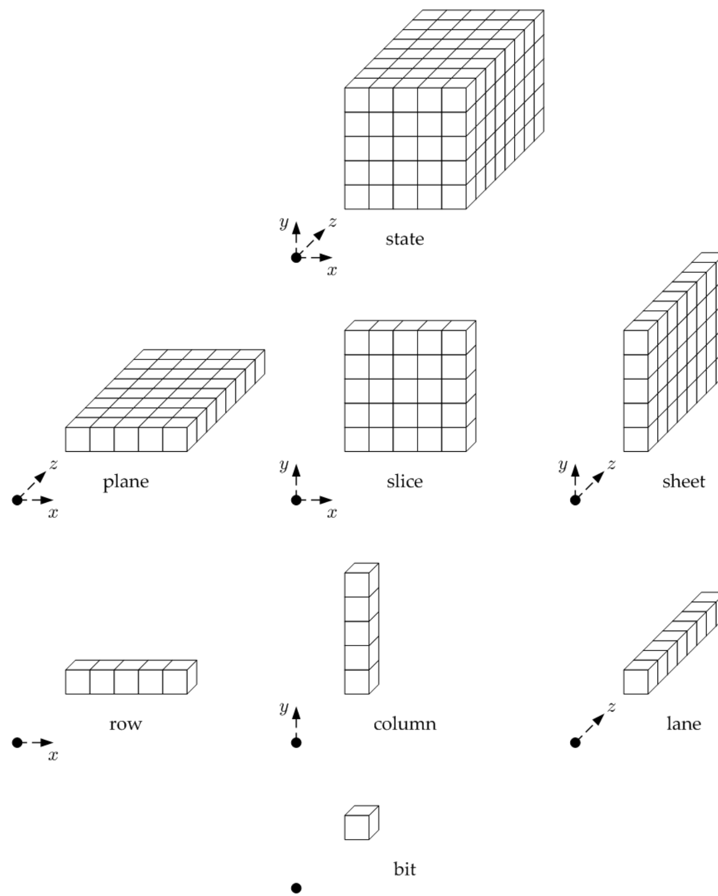


Figure 4-5: Naming conventions for the KECCAK *state* [24]

4.1.4.1 First step: theta

For every bit with coordinates [x,y,z] in the state, the bitwise sum of two KECCAK columns, [x-1,.,z] and [x+1,.,z-1] is taken and added to that bit [24]. A visual representation is depicted below.

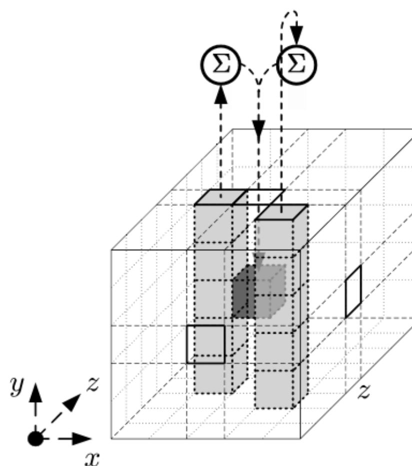


Figure 4-6: Theta [24]

4.1.4.2 Second step: rho

For each of the five sheets in the state, the bits from each column are translated in their lane by a number depending on the rho offsets [24]. These offsets are found in Table 4-2.

Table 4-2: Rho offsets [24]

	x = 0	x = 1	x = 2	x = 3	x = 4
y = 0	0	1	190	55	276
y = 1	36	300	6	55	276
y = 2	3	10	171	153	276
y = 3	150	45	15	21	136
y = 4	210	66	253	120	78

The translation values are defined by the offsets with modulo w. Figure 4-7 illustrates the Rho step.

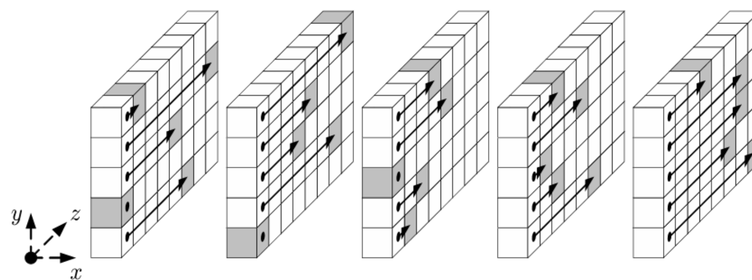


Figure 4-7: The rho step. Here x=0,y=0 is the centre of the middle sheet [24]

4.1.4.3 Third step: pi

In this step, all the lanes in each slice are shifted like so: x = y’ and y = 2x’+3y’, where x’ and y’ are the new positions. Within each slice, six axes can be formed: x axis, y axis, y=x axis, , y= -x axis, y=2x axis and the y =-2x axis [24]. The bits on a certain axis are translated to a different axis as depicted in Figure 4-8.

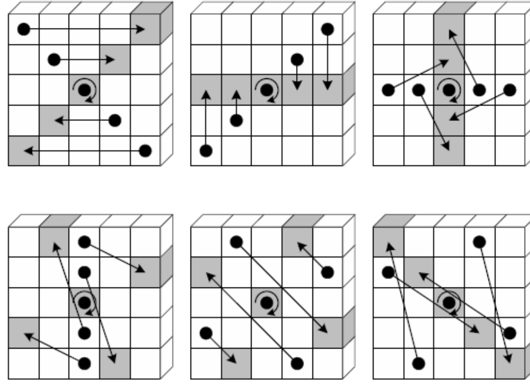


Figure 4-8: The pi step [24]

4.1.4.4 Fourth step: chi

In this step, for each bit [x] in every row, the logic AND between bit [x+2] and the inverse of bit [x+1] in the same row, are added to said bit [24]. This step is illustrated for a single row in Figure 4-9.

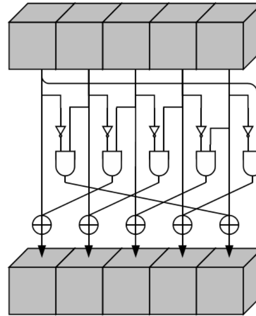


Figure 4-9: The chi step [24]

4.1.4.5 Fifth step: iota

In the final step, the round constant RC is added to the state. This RC is given by Equation 4-2.

$$RC[i_r]\left[0,0,2^j - 1\right] = rc[j + 7 \cdot i_r]\; for\; all\; 0\; \leq j \leq l$$

Equation 4-2: Round constant value [23]

With $l = 3$ in KetjeJr and $l = 4$ in KetjeSr. i_r is equal to the current round number plus 18 for KetjeJr and plus 20 for KetjeSr. It can be observed that only the bottom lane will get influenced by this sum, since the rest of RC is zero. The value of *rc[t]* is the output of a binary linear feedback shift register [23].

$$rc[t] = (x^t\; mod\; x^8 + x^6 + x^5 + x^4 + 1) mod\; x$$

Equation 4-3: rc constant value [23]

The table of round constants used in Ketje can be found in Table 4-3.

Table 4-3: Round constants for each of the modes in MonkeyDuplex

Start	Stride	Step	KetjeJr (hex)	KetjeSr (hex)
Start 1			0x81	0x008A
Start 2			0x09	0x0088
Start 3			0x8A	0x8009
Start 4			0x88	0x000A
Start 5			0x09	0x808B
Start 6			0x0A	0x008B
Start 7	Stride 1		0x8B	0x8089
Start 8	Stride 2		0x8B	0x8003
Start 9	Stride 3		0x89	0x8002
Start 10	Stride 4		0x03	0x0080
Start 11	Stride 5		0x02	0x800A
Start 12	Stride 6	Step1	0x80	0x000A

4.2 VHDL code

In this section, the design of the VHDL code is outlined. Just like with Trivia-ck, the data path and controller are separated to ease bug fixing. The top level block combines the inputs and outputs of these blocks together. The two subcomponents, the data path and the state machine are examined, as well as their underlying components.

4.2.1 Data path

The data path is built up out of four important blocks: the multiplexer controlled by the state machine, a counter, the keypack block, and the KECCAK round block.

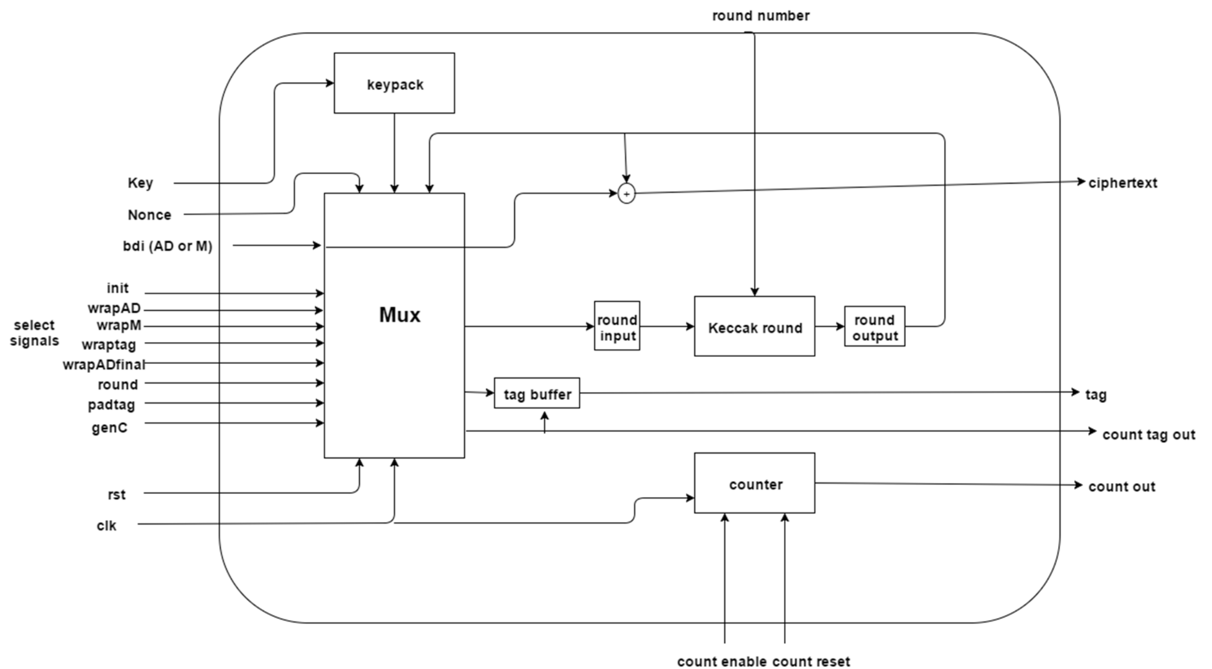


Figure 4-10: Ketje data path

The multiplexer's main function is the routing and padding of inputs as required by the MonkeyDuplex and MonkeyWrap constructs. The multiplexer has ten select bits originating from the state machine. The *init* input appends the keypack output with the nonce, pads it to match the state width b and routes it to the *round input* register. This register is used as the input to the KECCAK round block. The inputs *wrapAD, wrapADfinal, wrapM* and *wrapTag* all pad the input (AD, message or ciphertext) with the appropriate frame bits and with 10*1 padding and routes the padded input to the *I* register. The *padTag* input routes a padded "0" to the *round input* register. If *wrapM* is high, the ciphertext is also generated. If *padTag* is high, the tag is generated per sixteen bits in KetjeJr and per 32 bits in KetjeSr. The input *inccounter* increments a counter that keeps track of the padTag rounds. The amount of padTag rounds needed to generate a full tag is six for KetjeJr and four for KetjeSr.

The second block, the counter, keeps track of the KECCAK round number.

The keypack block pads the key with the bitwise keypack length and 10* padding, as previously discussed.

The last block, KECCAK round, incorporates the KECCAK-p permutation. How this block operates is highly dependent on the optimisation path. The speed optimised version calculates a complete round every clock cycle, while the area optimised version uses a memory block to store the state and calculates each step in the round lane per lane. These versions are discussed in depth in Sections 4.3.1 and 4.3.2.

It is also important to note that in order for the VHDL code to match the C source code, several extra functions are implemented. First is a function that reverses a string byte by byte, meaning the first byte becomes the last and so on. This function is called at the state input and output. The reason for this is that in the C source code the state's most significant byte is at position zero, while in VHDL this is at highest position. The second function has to do with extra padding. Since the pre-processor cannot pad the input as needed in the algorithm, it is done in the ciphercore. First the input is padded with zeros in

the pre-processor. This is needed because the ciphercore expects a 16-bit or 32-bit input. The pre-processor signals the ciphercore that the input has been padded, after which the padding function is called, which takes the valid bits out of the block and applies the right padding afterwards.

4.2.2 State machine

The state machine communicates with the pre- and post-processor and controls the select signals of the data path's multiplexer. Since the way data is processed depends highly on the KECCAK round block, the state machines differ for each optimisation path. The state machines follow the typical AEAD state structure, which begins with loading the key and npub into the state, processing the AD, processing the message or ciphertext and generating the tag. The two different versions of the state machines are discussed in Sections 4.3.1 and 4.3.2.

4.3 Ketje optimisation

Just as with Trivia-ck, the code is improved by using two optimisation stategies, area and speed. In total, three optimisations are performed. The first two optimisations are the different implementations of the KECCAK round block. Since this block is responsible for the state permutations, it is an obvious target for optimization. For both optimisations, the developer's VHDL code for the KECCAK hash function is used in the design of the round blocks. For the speed version, the KECCAK implementation "High speed core" is altered to fit with Ketje's design, and for the area version, the "low area co-processor" is chosen [25]. The third optimisation involves grouping several states in the state machine. This results in fewer states, which reduce the amount of slices and also increase clock speed. Sections 4.3.1 and 4.3.2 outline the area and speed optimised KECCAK round blocks, Section 4.3.3 discusses the new state machine and Section 4.3.4 outlines the final results with the improved state machine.

4.3.1 Area optimisation

The area optimised Ketje implementation uses system memory to store the state instead of having it stored internally. The KECCAK round block here has a data path and FSM of its own. The data path has three w-bit registers that are used to store temporary values. It also includes the blocks that preform the rho, chi and iota step. The state is read from the memory lane by lane. The FSM communicates with the memory to read or write the data on a specific address. The FSM also controls the register inputs via the command signal. In Figure 4-11, the KECCAK round block, here called coprocessor, is illustrated.

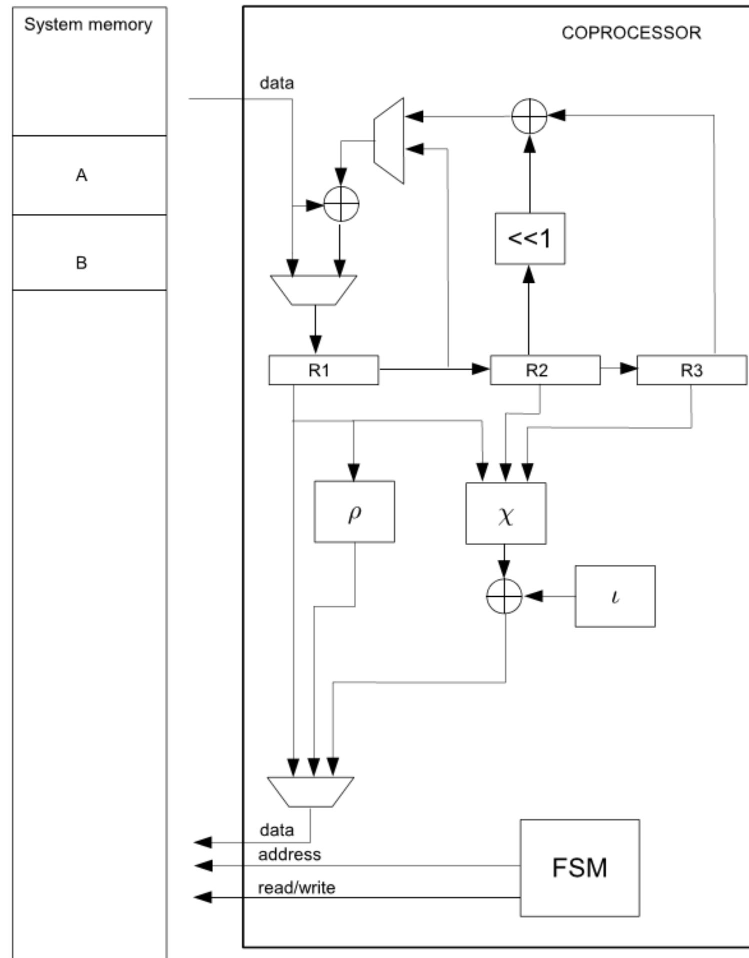


Figure 4-11: Area optimised K*ECCAK round block* [25]

A KECCAK round is performed in the following way. First the internal round number is initialised with the required start number. Now, the state get read lane by lane. These values are stored internally in register one. Each time a new lane is read, it gets xored with the previous stored value. This way, the sumsheet, required in theta, is calculated. After five lanes are read and xored, the resulting sumsheet is written to a dedicated area on the memory. This group of states is repeated until five sumsheets have been calculated.

In the next states, theta, pi and rho are calculated. First, the two sumsheets required for theta are read from the memory and stored in register two and three. Then, a lane is read, and in the following two states, the sum between this lane and the two sumsheets is calculated. This output is then shifted by the rho-offset and written to a specific area in the memory dictated by pi. This process gets repeated until all the lanes from the state have been read.

The following states calculate chi and iota. To compute chi, three lanes are read and stored in the registers, after which the chi block calculates the output. This repeats until all the lanes are read. Only this first output gets xored with iota's output, the others are send directly to the memory. This is because only the bottom lane gets influenced by the round constant. Now a full round has been completed. When the required number of rounds is computed, the FSM signals the top level state machine. The computation of one round takes 215 clock cycles.

Since this version of the code stores the state inside a memory block, several dedicated memory read/write states are implemented in the state machine. After the key and nonce are received, they are written to the memory. This whole process takes 25 clock cycles (200/8 or 400/16). During the state initialisation, the state is permutated by preforming 12 rounds. Since one round takes 215 cycles, this process takes a total of 2580 clock cycles.

The following states process the AD. First the AD gets padded. Then the state gets read from the memory, which is then xored with the padded AD and written back to the memory. Since the padded AD is always p+4 bits long, only the first p+4 bits of the state are read and written back to the memory, which saves a lot of time. The process of reading and storing this part of the state in a register takes five clock cycles. It takes two clock pulses for the data in the memory to get outputted. The other three cycles write this data to a register. The process of writing the data to the memory takes three clock cycles. Hereafter, the state is updated for one round. The process of reading the state, xoring it with the AD, writing the result back to the memory and updating repeats until all the AD is read.

The message is processed very similar, the one difference being that after reading the state, the ciphertext is generated by xoring it with the current message block.

During tag generation a new series of events commences consisting of: reading the state, generating a piece of the tag, xoring the state with a padded “0” bit, writing the result to the memory and preforming one round. An internal counter is used to determine the number of steps needed to generate the complete tag, which is outputted in the final state. The used state machine is illustrated below.

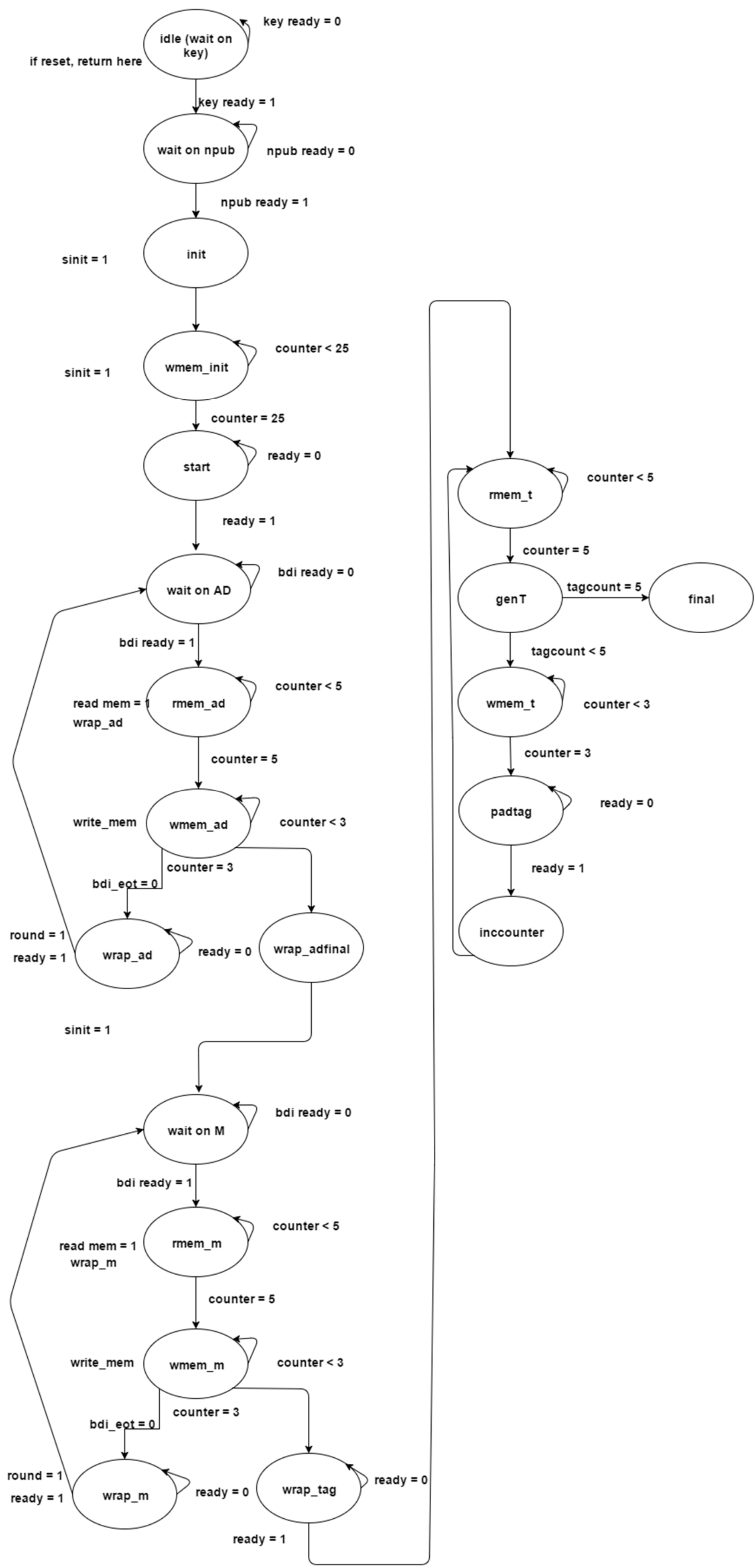


Figure 4-12: Ketje area state machine

4.3.2 Speed optimisation

In this version, the KECCAK round operates in a stand-alone fashion. The state input string gets converted onto a 3d- array by using Equation 4-1. Then the entire round gets calculated in a single clock cycle. All the bits in the state are transmuted all at once. First, the sumsheets are calculated, which are xored with the right state bits to calculate theta. After this, the lane bits are shifted by an amount equal to the rho offsets. Following the rho step, the state slices are shifted in the pi step. Hereafter, in the chi step, the rows are xored with one another. The final step, iota, xors the current round constant with the first lane. Then the output is converted back to a string and is stored in an output register.

This version of the state machine is very similar to the one used in the area optimised one. The different phases now require less steps to complete and the states used to write and read the memory are obviously not required here. The state machine is depicted in Figure 4-13.

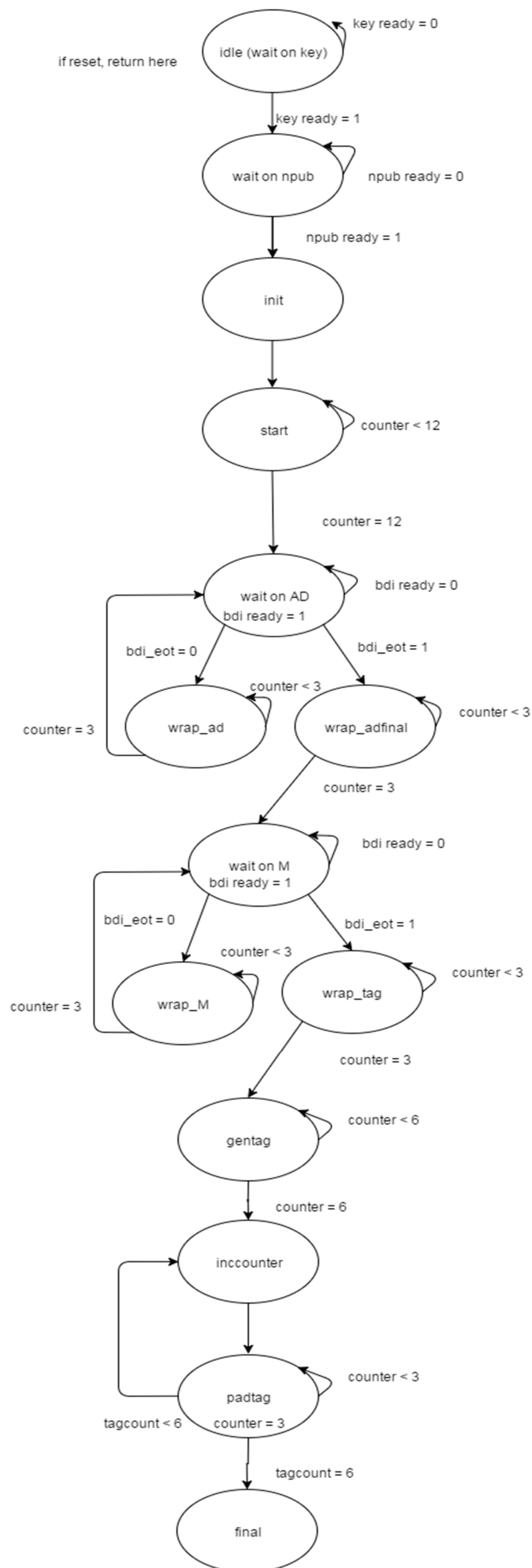


Figure 4-13: Ketje speed state machine

4.3.3 State machine optimisations

After reviewing the state machines, a lot of similarities can be found between the different states. *The wrapAD, wrapFinalAD, wrapM* and *wraptag* states all preform the same task: padding and loading their input into the state. The only difference between these states are the framebits.

In the area optimised state machine, three different states of *write* and *read* memory are used, while they all are identical. In this new state machine, the *wrap* states are all combined into a total of two states, *wrap* and *step*.

In the *wrap* state, the input gets padded and loaded into the state. The state machine now sends the framebits to the data path.

The *step* state performs the KECCAK round. This change removes four states in the speed optimised version and six in the area optimised version.

The tag generation states are unaltered, since these differ too much from the *wrap* states.

Losing these states also reduces the amount of signals to the data path multiplexer, reducing the select signal and thus the amount of different options. The downside to this state joining is that the state transitions become more complex. The results show that both the area is reduced and the throughput is increased. Figures 4-14 and 4-15 show both altered state machines.

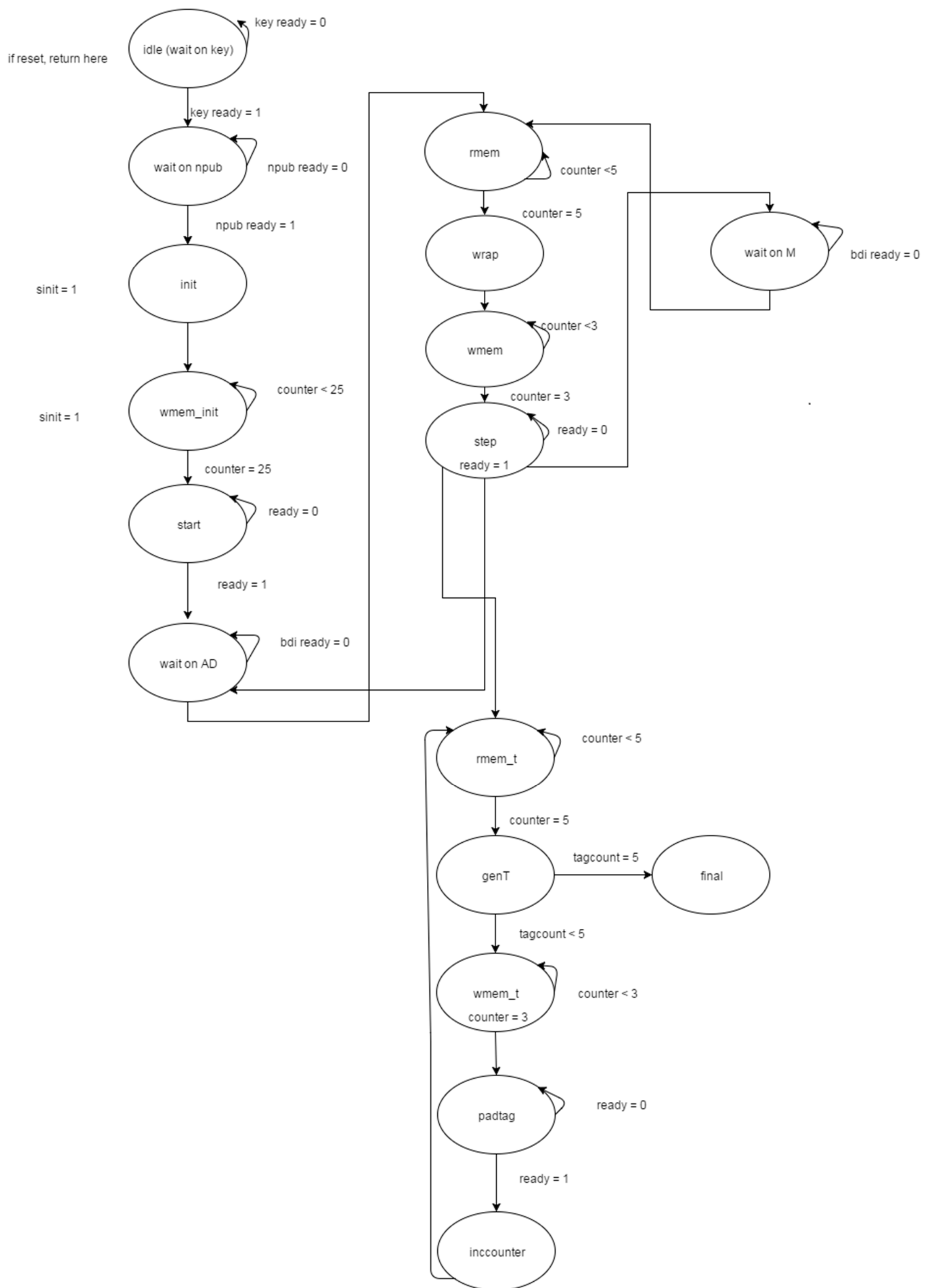


Figure 4-14: Area optimised Ketje with less states.

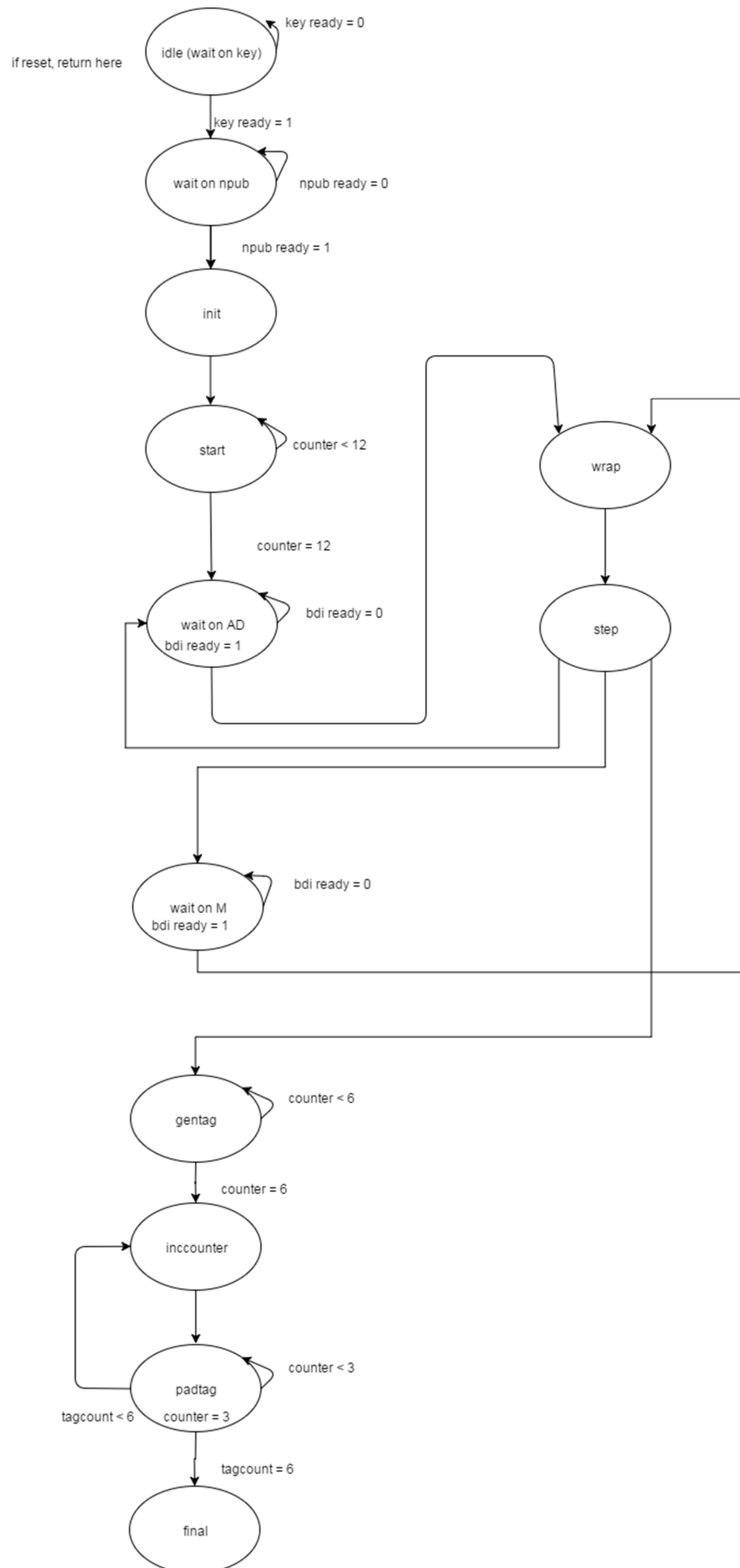


Figure 4-15: Speed optimised Ketje with less states

4.3.4 Results

In order to show how the optimised state machine influences the area and speed, the results for KetjeJr speed, KetjeJr area and KetjeSr speed are shown with and without this optimised state machine. The tables below show the results on area usage.

Table 4-4: Area results for speed optimised KetjeJr without/with optimised state machine

Device	LUTS	SLICES	FFs
Spartan 3	1137 / 1291	819 / 993	1252 / 1038
Spartan 6	1007 / 713	341 / 263	1132 / 887
Virtex 5	1173 / 764	400 / 288	1125 / 908
Virtex 6	954 / 707	310/ 237	1124 / 884

Table 4-5: Area results for area optimised KetjeJr without/with optimised state machine

Device	LUTS	SLICES	FFs
Spartan 3	947 / 940	751 / 745	934 / 932
Spartan 6	917 / 866	256 / 256	792 / 792
Virtex 5	776 / 775	327 / 315	802 / 800
Virtex 6	877 / 806	346 / 237	790 / 790

Table 4-6: Area results for speed optimised KetjeSr without/with optimised state machine

Device	LUTS	SLICES	FFs
Spartan 3	2106 / 2000	1405 / 1374	2116 / 1651
Spartan 6	1702 / 1322	443 / 404	1956 / 1464
Virtex 5	2168 / 1345	573 / 489	1953 / 1483
Virtex 6	1623 / 1327	514 / 375	1952 / 1459

It can be observed that in most cases the state machine with less states results in less LUT, slice and FF usage. Therefore the KetjeSr area optimised version was immediately implemented using this state machine. The results are found below.

Table 4-7: Area results for area optimised KetjeSr with optimised state machine

Device	LUTS	SLICES	FFs
Spartan 3	1389	1067	1257
Spartan 6	1204	322	1096
Virtex 5	1046	422	1104
Virtex 6	1133	433	1094

When comparing the Area optimised version to the speed optimised version of KetjeJr, the area is actually smaller or equal for the speed optimised version on the Virtex 5 and 6 family. In this case, it seems that the state size (200 bits) is too small to have a real advantage when stored in memory. The

coprocessor is based on VHDL code for KECCAK, which has an internal state of 1600 bits, and here, using memory will have a higher influence [25]. For KetjeSr, the results for the area optimisation are better, but not by a whole lot.

The results for the timing can be found in the tables below. Here, the two versions of each optimisation are compared again, except for KetjeSr area, which was only implemented with the optimised state machine, since the increase in throughput gained from the optimised state machine was apparent.

Table 4-8: Timing results for speed optimised KetjeJr without/with optimised state machine

Device	PAR Freq (MHz)	Throughput (Mbit/s)
Spartan 3	119,02 / 120,58	476,19 / 642,57
Spartan 6	126,79 / 127,32	507,16 / 679,06
Virtex 5	171,06/ 170,24	684,24 / 907,95
Virtex 6	320/ 210,57	1280/ 1122,81

Table 4-9: Timing results for area optimised KetjeJr without/with optimised state machine

Device	PAR Freq (MHz)	Throughput (Mbit/s)
Spartan 3	120,44 / 121,79	8,60 / 8,66
Spartan 6	130,62 / 171,20	9,33 / 10,46
Virtex 5	197,98/ 190,48	14,14 / 13,61
Virtex 6	336,36 / 311,72	24,03 / 22,86

Table 4-10: Timing results for speed optimised KetjeSr without/with optimised state machine

Device	PAR Freq (MHz)	Throughput (Mbit/s)
Spartan 3	93,61 / 98,32	749,06 / 1048,73
Spartan 6	126,81 / 126,92	1014,46 / 1353,81
Virtex 5	149,37 / 152,84	1194,92 / 1630,24
Virtex 6	202,63 / 210,35	1621,07 / 2243,72

Table 4-11: Timing results for area optimised KetjeSr with optimised state machine

Device	PAR Freq (MHz)	Throughput (Mbit/s)
Spartan 3	108,67	15,46
Spartan 6	126,82	18,04
Virtex 5	172,03	27,09
Virtex 6	240,04	44,33

As can be observed, the optimised state machine results in higher throughput in all cases. In some of the speed optimisations, the PAR frequency is actually lowered, however since the state machine requires one less state to compute the ciphertext, the throughput is still higher. In the area optimised versions, the state machine takes one extra clock cycle to calculate the ciphertext, but since the amount of cycles is already high, this has little influence. The amount of cycles needed to compute the ciphertext are three and 224 for the speed and area optimised version respectively without the

optimised state machine, and two and 225 with the optimised state machine. Using Equation 3-5, the calculation for the total encryption delay for area and speed versions without the optimised state machine are shown below.

$$encryption\ delay\ area(cycles) = 4801 + 224\frac{Mlen}{16} + 224\frac{ADlen}{16}\ or\ 4576 + 224\frac{Mlen}{32} + 224\frac{ADlen}{32}$$

$$encryption\ delay\ speed(cycles) = 30 + 4\frac{Mlen}{16} + 4\frac{ADlen}{16}\ or\ 27 + 4\frac{Mlen}{32} + 4\frac{ADlen}{32}$$

The constants are the cycles needed to load the key and nonce (one cycle), initialise the state (12 rounds), and finalise the tag (six rounds, plus another five for KetjeJr and another three for KetjeSr). With each extra round there is also an extra cycle needed to increment the counter. For the versions with optimised state machine for formulas stay the same, only the cycles for AD and M change.

$$encryption\ delay\ area(cycles) = 4801 + 225\frac{Mlen}{16} + 225\frac{ADlen}{16}\ or\ 4576 + 225\frac{Mlen}{32} + 225\frac{ADlen}{32}$$

$$encryption\ delay\ speed(cycles) = 30 + 3\frac{Mlen}{16} + 3\frac{ADlen}{16}\ or\ 27 + 3\frac{Mlen}{32} + 3\frac{ADlen}{32}$$

Shown in the table below are the latency results for the state optimised versions.

Table 4-12: Latency results for Ketje

Device	Latency (ns) KetjeJr area	Latency (ns) KetjeSr area	Latency (ns) KetjeJr speed	Latency (ns) KetjeSr speed
Spartan 3	39421 + 1839* (AD+M)	42108 + 2070 *(AD + M)	249 + 25 * (AD + M)	288 + 32 * (AD + M)
Spartan 6	32647+1530*(AD +M)	36082 + 1774 * (AD + M)	236 + 24 * (AD + M)	213 + 24 * (AD + M)
Virtex 5	25205 + 1176 *(AD + M)	24024 + 1181 *(AD + M)	176 + 18 * (AD + M)	181 + 20 * (AD + M)
Virtex 6	15402+719*(AD + M)	14680 + 722 * (AD + M)	143 + 14 * (AD + M)	133 + 15 * (AD + M)

It should be noted that the message and AD blocks are double the size in KetjeSr. Although it might seem that KetjeJr in some cases has a lower latency, for large inputs this is not the case, since KetjeJr will require twice as many AD and/or message processing cycles.

The speed optimised versions have a higher throughput and lower latency in all cases, up to 50 times higher in the case of throughput. Combining this with the observation that the speed optimised version for KetjeJr uses less area then the area version, it is concluded that the speed optimised version is superior. In the case of KetjeSr, the area optimised version offers only a small drop in area while suffering a high drop in throughput and increased latency when compared to the speed optimised version. So here it can also be concluded that the speed optimised version is superior.

Shown in Table 4-13 are the ASIC results using the NanGate PDK 45nm library. Again only the versions with an optimised state machine have been tested.

Table 4-13: ASIC results for Ketje implementations with optimised state machine

	Area (µm²)	Area (kGE)	Timing crit path (ns)	Max Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
KetjeJr Speed	6515	8,1	2,38	420,17	3361,34	71 + 7 * (AD + M)
KetjeJr Area	6250	7,8	2,37	421,94	30,00	11378 + 533 * (AD + M)
KetjeSr speed	10885	13,6	2,62	381,68	6106,87	71 + 8* (AD + M)
KetjeSr Area	9302	11,6	2,38	420,17	59,76	10891 + 536 * (AD + M)

Finally it should be noted that the AEAD interface does not work with KetjeJr, where it seems to have problems with its tag and npub size. In order to test the outputs, a simple wrapper is made, imitating the steps of the AEAD interface. KetjeSr however does work with the interface.

5 MORUS

5.1 Algorithm

The last tested algorithm, MORUS, was designed to be very fast in hardware. This is achieved by using only shifts, AND and XOR operations in the cipher [26]. There are three versions of MORUS: MORUS-1280-128, MORUS-640-128 and MORUS 1280-256. Their parameters can be found in the table below.

Table 5-1: MORUS parameters [26]

Recommended versions	Key (bits)	Nonce (bits)	State (bits)	Block length (bits)	Tag (bits)
MORUS-640-128	128	128	640	128	128
MORUS-1280-128	128	128	1280	256	128
MORUS-1280-256	256	128	1280	256	128

This thesis only handles the hardware implementations of MORUS-640 and MORUS-1280-128, which are the primary and secondary recommendations. MORUS-1280-256 is identical to MORUS 1280-128 except for the key size, so the difference in area and speed between to the two is small.

MORUS only uses one subcomponent, the state update function. This function is discussed in Section 5.1.1. The state of MORUS is divided into five blocks of identical length. In MORUS-640-128 (called MORUS-640 from this point on) the length of these blocks is 128 bits and in MORUS-1280-128 (called MORUS-1280 from this point on) this is 256 bits.

In the first step, initialisation, the key and npub are loaded into the state. In MORUS-640, the npub is loaded into the first state block and the key into the second. The third block is initialised with a 128-bit string of "1" and the fourth and fifth blocks are loaded with 128 bits of a 256-bit constant (the Fibonacci sequence modulo 256). In MORUS-1280, the npub is first padded to 256 bits with zeros before being loaded into the first block. The key is duplicated and appended (key||key) and then loaded into the second block. The fourth block is loaded with zeros and the fifth block is loaded with the 256-bit constant. Then the state gets updated sixteen times with state input "0". Hereafter the second state block is xored with the key (or key||key in MORUS-1280) to finalise the initialisation.

After initialisation, the AD is read block per block, with blocklength shown in Table 5-1. If a block is smaller than this blocklength, it gets padded with zeros. Then the state is updated once with the AD block as state input. This repeats until all AD blocks are processed.

In this cipher, encryption and decryption are done in a slightly different way. During encryption, a block of ciphertext is generated by xoring the message block with the first state block, the left shifted second state block (96-bit shift in MORUS-640 and 192-bit shift in MORUS 1280) and the AND result of the third and fourth state block. The message block is then used as input to update the state. During decryption, the message is generated in the same way, however the resulting message block is then used to update the state, instead of using the ciphertext block.

The tag is generated after encryption or decryption by first xoring the fifth state block with the first and loading the result into the fifth block. The state input of the fourth block is xored with the 64-bit representations of the message and AD length (these two 64-bit strings are appended, and are also

padded to 256 bits for MORUS 1280). The state then gets updated eight times. After this, the tag is generated by xoring the second to fifth state blocks [26]. The full process is illustrated in Figure 5-1.

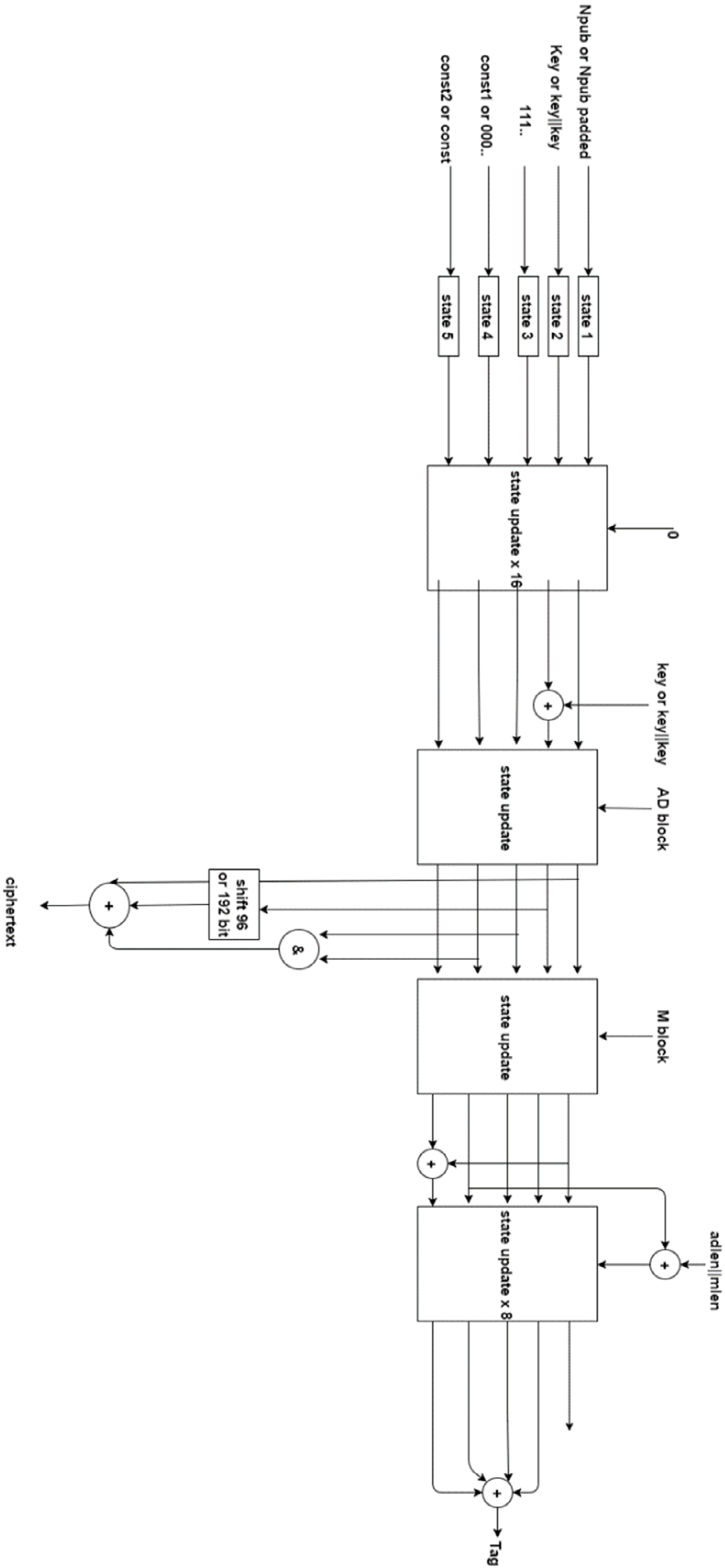


Figure 5-1: The encryption process of MORUS

5.1.1 State update

The state update function is responsible for updating the internal state. Its inputs are the five state blocks and a state input, which can be an AD block, message or ciphertext block, or "0", depending on the current phase. The state update function is built up out of five smaller rounds. Within each round, three different types of operations happen: rotate, shift and pass [26].

The rotate operation takes four state blocks and the state input (the state input is not used in the first round). The AND result of two of the state blocks is xored with the rest of the inputs. This results then gets divided into four words of 32 bits (in MORUS-640) or 64 bits (in MORUS-1280). Each of these words is then left shifted b bits. The value of b depends on the current round and can be found is Table 5-2.

Table 5-2: Rotation constants of MORUS [26]

b	MORUS-640	MORUS-1280
Round 1	5	13
Round 2	31	46
Round 3	7	38
Round 4	22	7
Round 5	13	4

The shift operation left shifts a state block by w bits. The values for w can be found in the table below.

Table 5-3: Shift constants of MORUS [26]

w	MORUS-640	MORUS-1280
Round 1	32	64
Round 2	64	128
Round 3	96	192
Round 4	64	128
Round 5	32	64

The pass operation simply passes the state block value to a new state block. The state update function is depicted below. The pseudocode for the function can be found in the appendix.

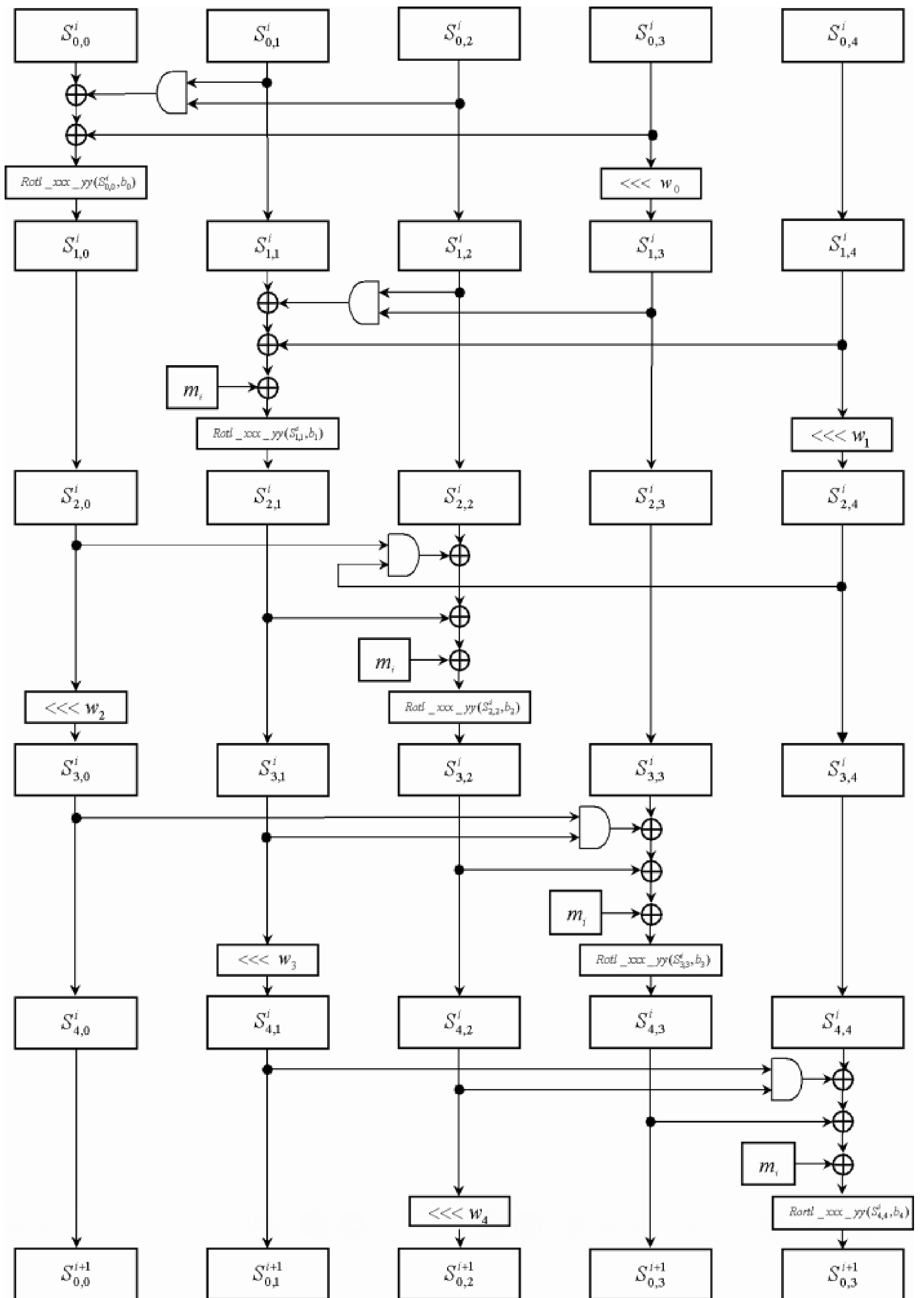


Figure 5-2: The state update function [26]

5.2 VHDL code

In this section, the design of the VHDL code is outlined. Both subcomponents: the data path and the state machine are examined, as well as their underlying components. As with the other algorithms, the top level block connects these two together.

5.2.1 Data path

The MORUS data path consists of three subcomponents. A multiplexer, responsible for routing the signals, a counter, responsible for counting the number of state update rounds and the state update block, responsible for updating the state. The data path is depicted below.

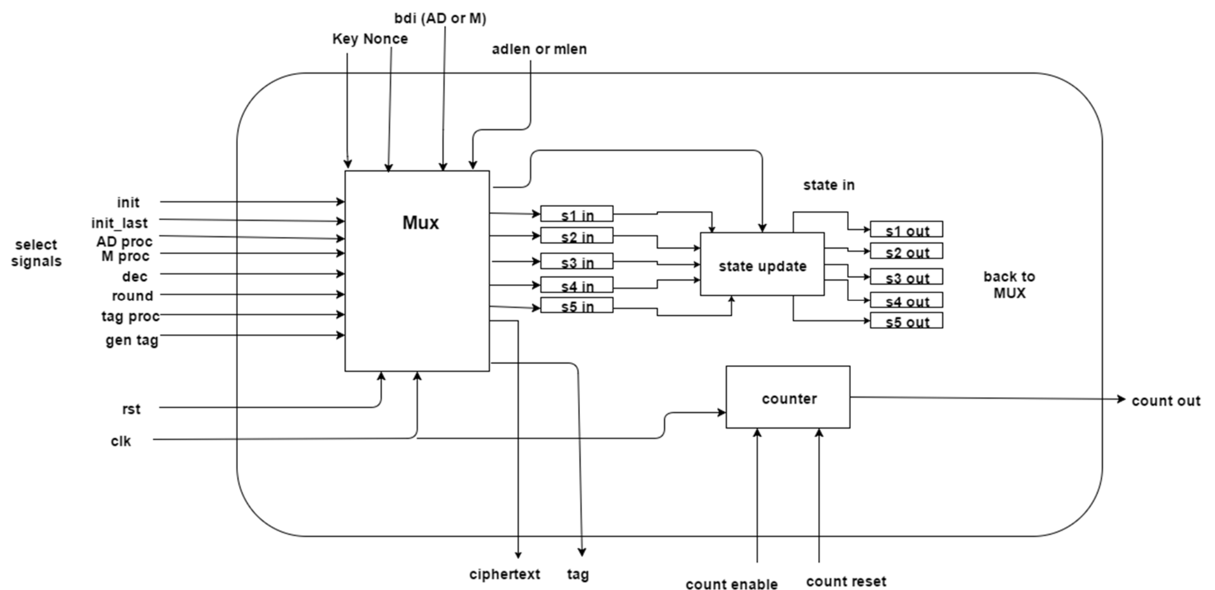


Figure 5-3: MORUS data path

The multiplexer's main function is routing the right inputs to the five state blocks. It is controlled by the state machine through its select signals. The multiplexer has eight select bits. The *init* input loads the key, npub and the constants into the state, as discussed previously. The *round* input routes the state update output back to its input. The counter counts how many rounds are needed. *Intit_last* is an input used after the sixteen initialisation rounds to perform the last initialisation step. When *AD_s* or *M_s* are high, the input blocks are first reversed and then loaded into the state. This reversing is needed because otherwise the results from the developer c-code do not match the VHDL results. This is because of the way arrays work in C, as discussed in Section 4.2. Also when one of these two inputs is high, the AD or message length is stored in a register. The pre-processor calculates these lengths, but the values are erased after the whole AD or message has been read, which requires them to be stored. Furthermore, when *M_s* is high, the ciphertext is generated. The *dec* input is used during decryption. It simply loads the generated message block into the state. *Tag_s* and *get_t* are used in the final stage of encryption or decryption. *Tag_s* is used to load the state with the message and AD lengths and updating the fifth state block like previously discussed. *Gen_t* generates the tag after eight round updates.

The state update block is divided into five “round” blocks. Each of these blocks alter the state blocks in a specific way, illustrated in Figure 5-2. The round outputs the five state blocks and takes as input the five blocks. All rounds except the first also use the state input. A full round is calculated over a single clock cycle.

5.2.2 State machine

The state machine is responsible for communicating with the AEAD interface and controls the data path multiplexer. Just like with Trivia-ck and Ketje, the MORUS state machine also has the distinct AEAD state groups. It waits on the pre-processor for the key and npub before beginning the initialisation. After loading the key and npub, 16 update rounds are performed. An extra state *init_last_s* updates the second state block. Then the state machine processes the AD. First the AD is loaded into the state and then one round is performed. The message processing is similar, *write_c* writes the output to the post-processor. After this, the tag is generated and written to the post-processor in the *final* state. The state machine is illustrated in Figure 5-4.

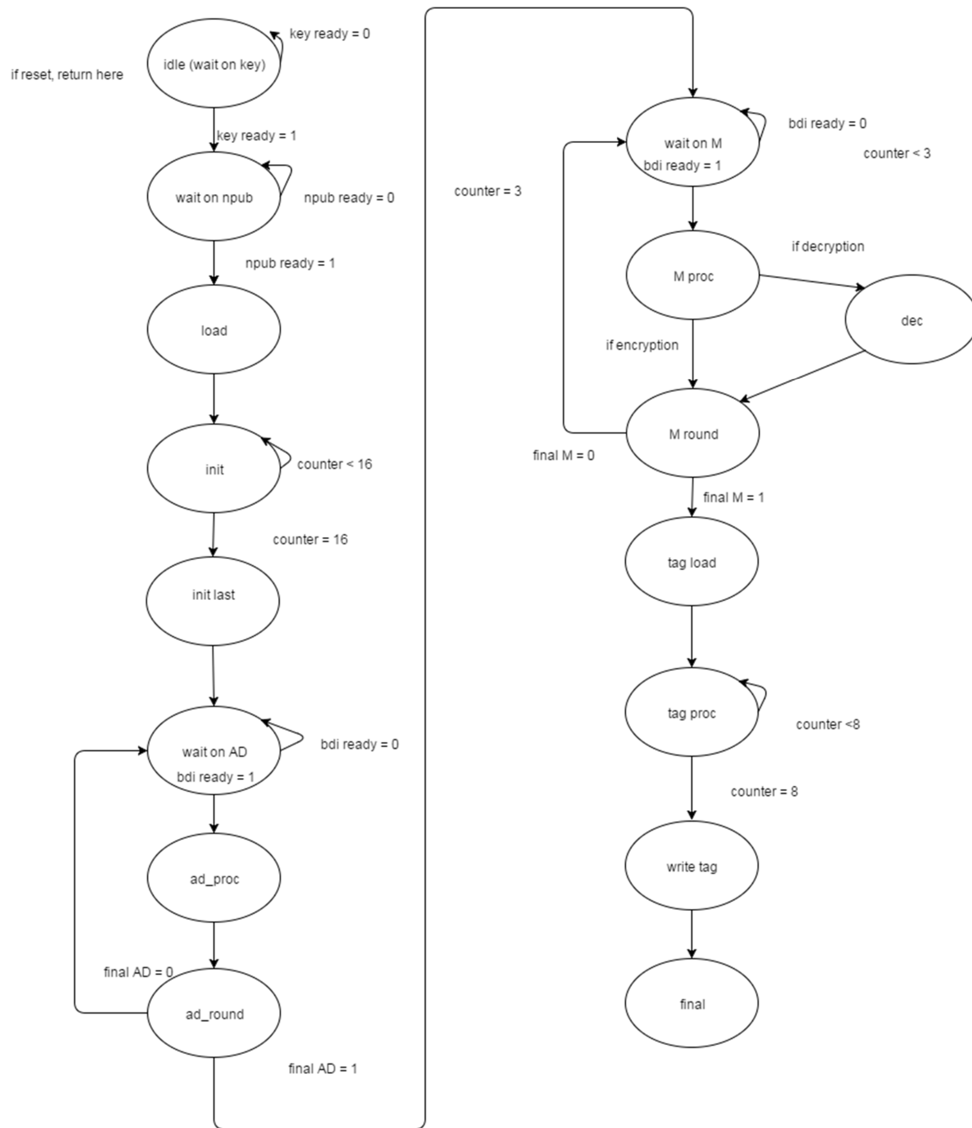


Figure 5-4: MORUS state machine

5.3 MORUS optimisation

The optimisation paths used for MORUS are the two variants, MORUS 640 and MORUS 1280. MORUS 640 has a smaller state, resulting in less hardware usage. MORUS 1280 has double the block size of MORUS 640, thus having higher throughput but using more area. Several other optimisations have been tested, but none improved the code.

The state update function is tested first. In the basic version, five steps are calculated at once, using 25 signals to connect them together. In order to reduce this, the state update function is reduced to having only one round block in a loop, similar to the Trivia-ck multiplier optimisation. However, whereas the FM blocks of Trivia-ck where completely identical, the round blocks are not. The first difference is the shift constants used in every round, the second one is that the first round block does not use the state input during its rotate function. Thus an extra controller is needed to load the right shift constants into the round loop at the right time, as well as requiring an extra rotate function to calculate the first round. This controller uses the data path counter as its select signal. And, although the area usage of the state update block was reduced, the controller almost doubled the total amount of slice LUTs and slice registers, so this implementation was undone. MORUS state update function also does not allow

for an obvious way to reduce the input size, compared to Trivia-ck, which uses the flexible Trivia-SC, and Ketje, which uses round blocks that can calculate their output lane per lane. In MORUS, each round uses all five state blocks in XOR and AND operations, and because shifting is used, it is not possible to reduce the state block sizes.

For speed optimisations, placing pipeline registers between the rounds was considered, but undone since the operations used here are already built for speed. The second reason is that the state update uses feedback. So pipelining would increase the latency considerably for every message or AD block, and thus would lower the throughput [27].

Since reducing the states in Ketje helped with both speed and area, the same was attempted here. However, Ketje's base state machine has several states that functioned identical, here this is not the case. Only the *ad_round* and *m_round* states are identical but use the same output signal so merging them would result in more complex state transitions, while not reducing the amount of select signals used in the data path. It is therefore concluded that the basic versions offer the best results for area and speed.

5.3.1 Results

Below the results for area can be found for MORUS-640, which is the area optimised version of the two implementations and MORUS-1280, which is the speed optimised version.

Table 5-4: Area results for MORUS-640.

Device	LUTS	SLICES	FFs
Spartan 3	3271	2234	2134
Spartan 6	2119	715	1986
Virtex 5	1980	567	1955
Virtex 6	1840	533	1815

Table 5-5: Area results for MORUS-1280.

Device	LUTS	SLICES	FFs
Spartan 3	5930	3641	3334
Spartan 6	3371	1175	3879
Virtex 5	3515	1190	3116
Virtex 6	3400	1081	3114

In the tables 5-6 and 5-7, the timing results can be found for both MORUS implementations.

Table 5-6: Timing results for MORUS-640

Device	PAR Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
Spartan 3	120,41	5137,47	233+25*(M + AD)
Spartan 6	127,31	5431,78	220+24*(M + AD)
Virtex 5	142,09	6062,33	197+21*(M + AD)
Virtex 6	284,25	12128	99+11*(M + AD)

Table 5-7: Timing results for MORUS-1280

Device	PAR Freq (MHz)	Throughput (Mbit/s)	Latency (ns)
Spartan 3	120,60	10291	233+25*(M + AD)
Spartan 6	147,54	12586	190+20*(M + AD)
Virtex 5	138,06	11781,50	203+22*(M + AD)
Virtex 6	202,14	17249,5	139+15*(M + AD)

The encryption delay for both implementations are shown below.

$$encryption\ delay\ (cycles) = 28 + 3\frac{Mlen}{128} + 3\frac{ADlen}{128}$$

The constant here contains the cycles needed to load the key and nonce (one cycle), initialise the state (16 rounds), and finalise the tag (eight rounds and two cycles, one to load the input and another to generate the tag). Each message or AD block takes three cycles to process. First, the input is loaded and the ciphertext is formed, then one round is performed, then the state waits for the next block.

Shown below are the ASIC results using the NanGate PDK 45nm library.

Table 5-8: ASIC results for MORUS implementations

	Area (μm^2)	Area (kGE)	Timing crit path (ns)	Max Freq (MHz)	Throughput (Mbit/s)	Latency(ns)
MORUS 640	27468	34,3	2,53	395,26	16864,3	71 + 8 * (AD + M)
MORUS 1280	58068	72,59	2,37	421,94	36005,6	66 + 7 * (AD + M)

6 Performance comparison

In this section, the results from the three implemented algorithms are compared to one another and are compared to other ciphers from the ATHENa database.

In the table below, the results of each algorithm's implementation using the lowest area on Virtex 6, are shown.

Table 6-1: Results for minimum area

Algorithm	LUTS	SLICES	FFs	Throughput (Mbit/s)	Latency (ns)
Trivia-ck (area optimised)	1724	699	2112	156,83	10635+408,1*(M + AD)/64
KetjeJr (speed optimised)	707	237	884	1122,81	143 + 14 * (AD + M)/16
MORUS 640	1840	533	1815	12128	99+11*(M + AD)/128

As expected, KetjeJr has the smallest hardware footprint, since it is an algorithm designed for area constrained devices. MORUS is second in area usage, but it has the highest throughput. Trivia-ck has the most slices and the lowest throughput, therefore it is not a very good implementation. The results for the other chips are similar, with the exception that the area optimised Trivia-ck uses less slices than MORUS 640 on Spartan 3 and 6. Trivia-ck uses 1593 slices on Spartan 3 and 618 on Spartan 6, while MORUS uses 2234 and 715 slices respectively. The throughput of MORUS is still considerably higher than that of Trivia-ck on both chips.

In the table below, the results of each algorithm's implementation with the highest throughput and lowest latency on Virtex 6, are shown.

Table 6-2: Results for maximum speed

Algorithm	LUTS	SLICES	FFs	Throughput(Mbit/s)	Latency(ns)
Trivia-ck (speed optimised)	3981	1205	6153	7522,24	463,7+9*(M+AD)/64
KetjeSr (speed optimised)	1327	375	1419	2243,72	133 + 15 * (M + AD)/32
MORUS 1280	3400	1081	3114	17249,5	139+15*(M + AD)/256

Here it can clearly be observed that MORUS has superior Throughput and the lowest Latency, while still having a smaller area footprint than Trivia-ck. Ketje, having the smallest footprint, also has the lowest throughput and high latency. Trivia-ck has the fastest message processing time, as can be seen in the latency. The results are similar on all chip families and are not shown here.

Finally the Throughput/area is calculated for each algorithm to indicate the overall area efficiency in Virtex 6.

Table 6-3: Throughput/ area results for both optimisation paths

Algorithm	Area efficiency (Mbs /slice)
Trivia-ck (area optimised)	0,22
Trivia-ck (speed optimised)	6,24
KetjeJr (speed optimised)	4,74
KetjeSr (speed optimised)	5,98
MORUS 640	22,75
MORUS 1280	15,96

As can be seen here, MORUS has the highest efficiency in both implementations. It can be concluded that the MORUS algorithm offers the most efficient and versatile hardware implementations, having the second lowest area but having the highest speed by far.

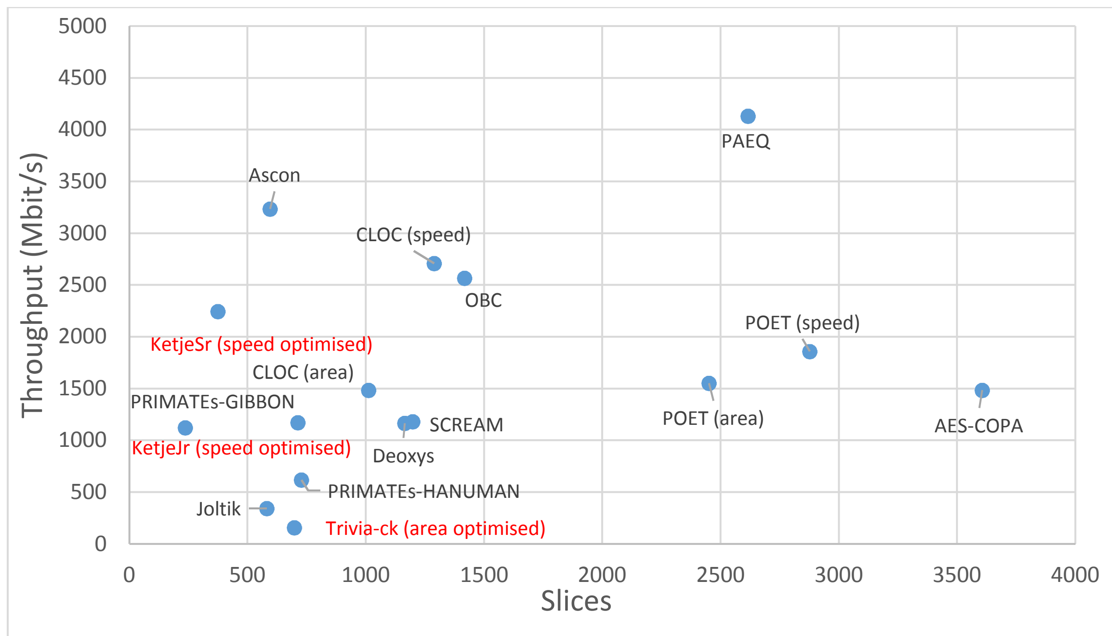
It should be mentioned that both Trivia-ck and MORUS have other hardware implementations [28], [29].The other Trivia-ck implementation uses 725 slices and has 16000 Mbit/s throughput in Virtex 6. It must be taken into account that the developers did not mention their API. When looking at their cycles per byte results, it shows that their ciphertext takes only one cycle to generate. In the AEAD version, it takes an extra clock cycle to wait on the pre-processor. This will make for different throughput values. The maximum frequency is not given, so it is unknown whether the post-PAR frequency or the synthesis frequency was used in calculating the throughput. The same goes for MORUS, where MORUS-640 has an area of 485 slices and a frequency of 425 MHz, resulting in 54400 Mbit/s throughput, and 879 slices, 370,4 MHz and 94,8 Gbit/s for MORUS 1280. Again, no information is given about the API and which frequency is used. Also the results are on a Virtex 7 chip, which also influences the results.

In order to have a more fair comparison, the algorithms are compared with other second round candidates that are also implemented with the AEAD structure. These results are taken from the ATHENa database [30].The table below shows the type, key size, Throughput, implementation frequency and the amount of slices of each cipher. All of the ciphers are tested on the Virtex 6 family. These algorithms have been optimised for good Throughput/area ratio, which is a balanced approach. There are no other implementations using AEAD for Trivia-ck, Ketje and MORUS yet. The results are shown in Table 6-4.

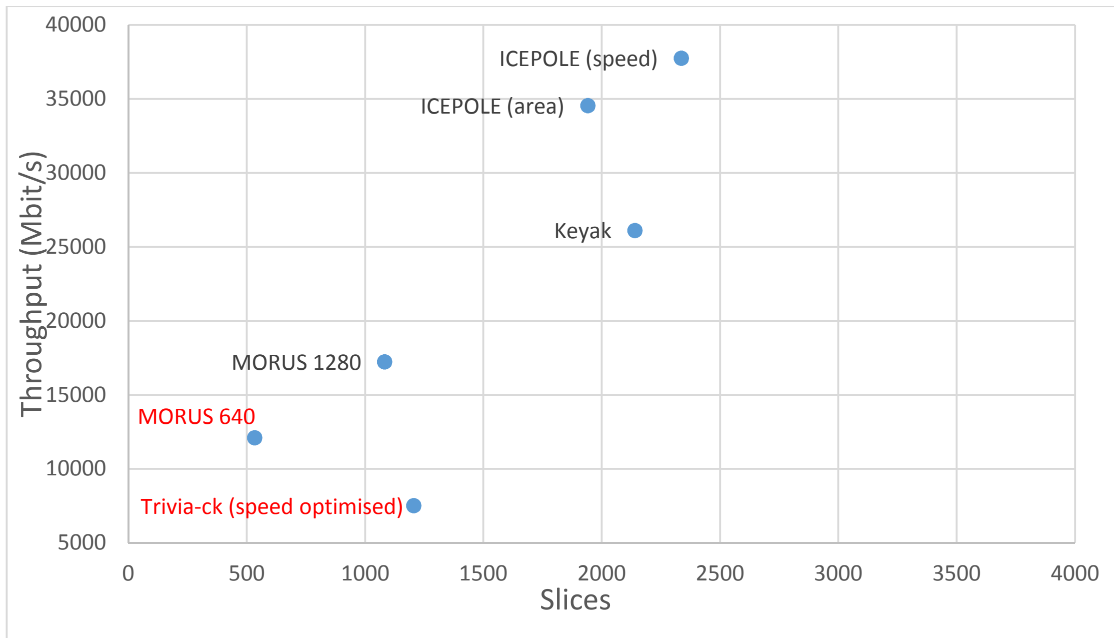
Table 6-4: CAESAR results implemented with the AEAD core on Virtex 6 [30]

Algorithm	Type	Key size (bits)	PAR Freq (MHz)	Throughput (Mbit/s)	Slices	Area efficiency (Mbs /slice)
AES-COPA	Block cipher	128	127,36	1482	3606	0,41
Ascon	Sponge	128	353,86	3235	595	5,44
CLOC (speed)	Block cipher	128	232,78	2709	1290	2,10
CLOC (area)	Block cipher	128	159,21	1482	1012	1,46
Deoxys	Block cipher	128	264,20	1166	1165	1,00
ICEPOLE (area)	Sponge	128	202,55	34569	1940	17,82
ICEPOLE (speed)	Sponge	128	258,26	37780	2336	16,17
Joltik	Block cipher	128	348,80	343	582	0,59
Keyak	Sponge	128	233,26	26126	2139	12,21
OBC	Block cipher	128	220,51	2566	1418	1,81
PAEQ	Block cipher	128	225,84	4130	2617	1,58
POET (area)	Block cipher	128	133,44	1553	2452	0,63
POET (speed)	Block cipher	128	159,80	1859	2877	0,41
PRIMATEs-GIBBON	Sponge	120	205,00	1171	713	5,44
PRIMATEs-HANUMAN	Sponge	120	200,40	617	728	2,10
SCREAM	Block cipher	128	101,33	1179	1199	1,46

In order to compare the algorithms, their throughput and slices have been plotted in the graph below. The algorithms have been divided into two categories, ones that have a throughput of less than 5000 Mbit/s and the others that have more. This is done in order to get more orderly graphs.



Graph 6-1: Throughput/ Area of CAESAR candidates (low Throughput)



Graph 6-2: Throughput/ Area of CAESAR candidates (high throughput)

As can be observed from the graphs, Ketje has the smallest area footprint of all the tested candidates, with both versions being the only algorithms that use less than 500 slices. Trivia-ck area implementation has the lowest throughput of all algorithms, but is in the top six in the speed implementation. MORUS has excellent results with both optimisations, being in the top five highest throughput candidates, while still having the lowest area usage. ICEPOLE, based on a KECCAK-like permutation and Keyak, based on KECCAK-f (just like Ketje) seem to perform really well. Ketje was designed for area, Keyak for speed, and both algorithms top the chart in their design choice. Therefore, the KECCAK permutation seems very sound to build efficient AE hardware implementations.

7 Conclusion & discussion

The main objective of this thesis was to implement and test three CAESAR ciphers that made it to the second round, on their area usage and speed. Trivia-ck, Ketje and MORUS have all been successfully implemented in VHDL for both minimal area and maximal speed separately. All the ciphers, except the KetjeJr variant, have been tested using the AEAD core API using various message and AD lengths. Using the ATHENa tool and Design compiler, area and timing results have been generated for all ciphers and finally used in a comparison with other ciphers using the AEAD API.

These results have shown that Ketje, an algorithm built for area constrained devices proved to be the smallest of all the ciphers, with a minimal area of 237 slices on Virtex 6. In order to further minimize the area for Ketje, the VHDL implementation of the KECCAK low area co-processor was used, but this resulted in more area, thus proving that the speed optimised KECCAK round is better suited for both optimisation paths. MORUS, having the second highest area efficiency of all the ciphers, proved to be the most versatile of the three tested ciphers, it having small area and high speed on both builds. Several optimisation techniques where attempted on MORUS, but none improved the results. The Trivia-ck area optimisation, while having 50% less area usage than the basic version, suffers from low throughput. The speed optimised version has good throughput results, but does not surpass MORUS and has more slice usage.

The main advantages MORUS and Ketje have over Trivia-ck is their simple round operations. Using simple XOR and shift operations cost very little in hardware, compared to the 32-bit multiplier used in Trivia-ck, using a 32-bit 2:1 multiplexer for each FM block. For the area optimised path, these FM blocks are reduced to two blocks, significantly reducing the area when compared to the speed optimised version, where 32 FM blocks are used. However, reducing this multiplier to only two FM blocks significantly increases the processing time of the AD, message and Tag. The reduction of the Trivia-SC inputs increased this even further. Since the goal of the optimisation was to limit the area, a more balanced approach, which could have resulted in better area efficiency, was not pursued.

Future work for this subject can be implementing the algorithms using a more balanced path, for instance, the Throughput/area path that is used in the other AEAD ciphers. Another feature that could be tested is maximising throughput with a fixed frequency. Since the maximum clock frequency is fixed on FPGA boards, it could be interesting to make a code that still has high throughput. This can for instance be achieved by using multiple round blocks of the cipher to calculate the ciphertext in fewer cycles. For Trivia-ck, more research could be used to design a more efficient multiplier. The current version is inspired by the developer C-code, and this might not be the most efficient version. Another optimisation for Trivia-ck can be changing the Trivia –SC cipher. Now only the 64-bit and 1-bit variants are tested, since these where the two recommended values. Further testing the range between the two values can perhaps result in better throughput/area results. The current versions of the algorithms are tested with their recommended parameters, like key size and npub size. Using the current implementations as a starting point, versions could be made where the user can choose the size of these parameters.

Literature

[1] R. Rivest, “Handbook of Theoretical Computer Science 1,” in *Handbook of Theoretical Computer Science 1*, 1990.

[2] “Cryptography/Introduction - Wikibooks, open books for an open world.” [Online]. Available: https://en.wikibooks.org/wiki/Cryptography/Introduction. [Accessed: 10-Jan-2016].

[3] “Authenticated Encryption - Crypto++ Wiki.” [Online]. Available: https://www.cryptopp.com/wiki/Authenticated_Encryption. [Accessed: 07-Jan-2016].

[4] “DES Cracker Project.” [Online]. Available: https://en.wikipedia.org/wiki/EFF_DES_cracker. [Accessed: 14-Jan-2016].

[5] “Crypto competitions: Introduction.” [Online]. Available: http://competitions.cr.yp.to/index.html. [Accessed: 07-Jan-2016].

[6] “Verilog vs. VHDL | BitWeenie.” [Online]. Available: http://www.bitweenie.com/listings/verilog-vs-vhdl/. [Accessed: 08-Jan-2016].

[7] “Algemene informatie – ESAT KU Leuven.” [Online]. Available: https://www.esat.kuleuven.be/info. [Accessed: 07-Jan-2016].

[8] “About us | COSIC.” [Online]. Available: http://www.esat.kuleuven.be/cosic/?page_id=13. [Accessed: 07-Jan-2016].

[9] S. Flushrer, I. Mantin, and A. Shamir, “Weaknesses in the Key Scheduling Algorithm of RC4,” *Sel. Areas Cryptogr.*, pp. 1–24, 2001.

[10] H. Krawczyk, “The order of encryption and authentication for protecting communications (or: How secure is SSL?),” *Adv. Cryptology—CRYPTO 2001*, vol. 2139, pp. 310–331, 2001.

[11] “Authenticated encryption - Wikipedia, the free encyclopedia.” [Online]. Available: https://en.wikipedia.org/wiki/File:Authenticated_Encryption_EtM.png. [Accessed: 07-Jan-2016].

[12] S. P. Mansoor, “Performance analysis of stream and block cipher algorithms,” in *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, 2010, vol. 1, pp. V1–522–V1–525.

[13] “Chapter 5. Wired Equivalent Privacy (WEP) - 802.11 Wireless Networks: The Definitive Guide, Second Edition.” [Online]. Available: http://flylib.com/books/en/2.519.1.37/1/. [Accessed: 07-Jan-2016].

[14] "The Keccak sponge function family." [Online]. Available: http://keccak.noekeon.org/. [Accessed: 07-Jan-2016].

[15] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. U. Sharif, and K. Gaj, "GMU Hardware API for Authenticated Ciphers," *Eprint.Iacr.Org*, 2015.

[16] K. Gaj, J. P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa - Automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs," *Proc. - 2010 Int. Conf. F. Program. Log. Appl. FPL 2010*, pp. 414–421, 2010.

[17] A. Chakraborti and M. Nandi, "TriviA-ck-v1," 2014.

[18] A. Chakraborti and M. Nandi, "TriviA-ck-v2," 2015.

[19] C. De Canniere and B. Preneel, "TRIVIUM Specifications," *ECRYPT Stream Cipher Proj. Rep.*, vol. 30, p. 2005, 2005.

[20] J. Stephenson, "Logic Optimization Techniques for Multiplexers," *Altera Lit.*, no. March, pp. 1–8, 2004.

[21] Model Technology, "Applications Note 116 : VHDL Style Guidelines for Performance."

[22] E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," no. 60, 2010.

[23] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "CAESAR submission : Ketje v1," 2014.

[24] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The keccak reference," *Submiss. to NIST (Round 3)*, pp. 1–14, 2011.

[25] B. Guido, "Keecak Implementation Overview," pp. 1–59, 2012.

[26] H. Wu and T. Huang, "MORUS v1," pp. 1–19, 2014.

[27] S. W. Alexander and R. W. Stewart, "The Effects of Pipelining Feedback Loops in High Speed DSP Systems," in *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, 2005, vol. 5, pp. 145–148.

[28] A. Chakraborti, A. Chattopadhyay, M. Hassan, and M. Nandi, "TriviA: A Fast and Secure Authenticated Encryption Scheme," pp. 1–26.

[29] H. Wu and T. Huang, "MORUS: A Fast Authenticated Cipher," 2015, pp. 1–23.

[30] "ATHENa Database of FPGA Results for Authenticated Ciphers." [Online]. Available: https://cryptography.gmu.edu/athenadb/fpga_auth_cipher/table_view. [Accessed: 07-Jan-2016].

Appendix

Developer Pseudocode Trivia-ck[17]

Load (K; IV): Key and IV Loading:

$(A_1,A_2,A_3, \dots A_{132}) \leftarrow (K_1,K_2,K_3, \dots,K_{128}, 1, 1, 1, 1)$;

$(C_1,C_2,C_3, \dots,C_{147}) \leftarrow (IV_1, IV_2, IV_3, \dots, IV_{128}, 1, 1, \dots 1)$;

$(B_1,B_2,B_3, \dots,B_{105}) \leftarrow (1, 1, 1, \dots, 1)$;

Update: Update a Single Round

$t_1 \leftarrow A_{66}$ xor A_{132} xor (A_{130} and A_{131}) xor B_{96};

$t_2 \leftarrow B_{69}$ xor B_{105} xor (B_{103} and B_{104}) xor C_{120};

$t_3 \leftarrow C_{66}$ xor C_{147} xor (C_{145} and C_{146}) xor A_{75};

$(A_1,A_2,A_3, \dots,A_{132}) \leftarrow (t_3,A_1,A_2, \dots,A_{131})$;

$(B_1,B_2,B_3, \dots,B_{105}) \leftarrow (t_1,B_1,B_2, \dots,B_{104})$;

$(C_1,C_2,C_3, \dots,C_{147}) \leftarrow (t_2,C_1,C_2, \dots,A_{146})$;

KeyExt : Extract a Key Bit

z = A_{66} xor A_{132} xor B_{69} xor B_{105} xor C_{66} xor C_{147} xor (A_{102} and B_{66}) ;

Output z ;

StExt64 : Extract 64 Key Bits

Output $A_1,A_2, \dots ,A_{64}$;

Insert (T) :Insert T into State Registers

$(S_1, S_2, \dots , S_T) = (S_1, S_2, \dots , S_T)$ xor T ;

KeyExt64: Extract First 64 Bits from A After 64 Rounds

t = $A_{[3\dots66]}$ xor $A_{[69\dots132]}$ xor $B_{[6\dots69]}$ xor $B_{[42\dots105]}$ xor $C_{[3\dots66]}$ xor $C_{[84\dots147]}$ xor

$A_{[39\dots102]}$ and $B_{[3\dots66]}$;

Output t ;

Update64: Update 64 Rounds

t1 A[3...66] xor A[69...132] xor A[67...130] and A[68...131] xor B[33...96] ;

t2 B[6...69] xor B[42...105] xor B[40...103] and B[41...104] xor C[57...120] ;

t3 C[3...66] xor C[84...147] xor C[82...145] and C[83...146] xor A[12...75] ;

(A1,A2,A3, ...,A132) ← (t3,A1,A2, ...,A68);

(B1,B2,B3, ...,B105) ← (t1,B1,B2, ...,B41) ;

(C1,C2,C3, ...,C147) ← (t2,C1,C2, ...,A83) ;

Developer Pseudocode KECCAK-f round[23]

$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$, with

Θ: $a[x, y, z] \leftarrow a[x, y, z] + \sum_{y=0}^{4} a[x-1, y, z] + \sum_{y=0}^{4} a[x+1, y, z-1]$

P: $a[x, y, z] \leftarrow a\left[x, y, z - \frac{(t+1)(t+2)}{2}\right]$ with t satisfying $0 \leq t < 24$ and $\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}^t \begin{matrix} 1 \\ 0 \end{matrix} = \begin{matrix} x \\ y \end{matrix}$ in $GF(5)^{2x2}$

or $t = -1$ if $x = y = 0$

π: $a[x, y] \leftarrow a[x', y']$, with $\begin{matrix} x \\ y \end{matrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{matrix} x' \\ y' \end{matrix}$

χ : $a[x] \leftarrow a[x] + (a[x+1] + 1)a[x+2]$

ι : $a \leftarrow a + RC[i_r]$

with $RC[i_r][0,0,2^j - 1] = rc[j + 7i_r]$ for all $0 \leq j \leq l$

with $rc[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x$ in $GF(2)[x]$

Developer Pseudocode MORUS state update [26]

S^{i+1} = StateUpdate(S^i,m_i) with 5 rounds

Round 1 : $S^i_{1,0}$ = Rotl_xxx_yy($S^i_{0,0}$ xor ($S^i_{0,1}$ and $S^i_{0,2}$) xor $S^i_{0,3}$, b_0);
$S^i_{1,3} = S^i_{0,3} \lll w_0$;
$S^i_{1,1} = S^i_{0,1}$;
$S^i_{1,2} = S^i_{0,2}$;
$S^i_{1,4} = S^i_{0,4}$;

Round 2 : $S^i_{2,1}$ = Rotl_xxx_yy($S^i_{1,1}$ xor ($S^i_{1,2}$ and $S^i_{1,3}$) xor $S^i_{1,4}$ xor m_i, b_1);
$S^i_{2,4} = S^i_{1,4} <<< w_1$;
$S^i_{2,0} = S^i_{1,0}$;
$S^i_{2,2} = S^i_{1,2}$;
$S^i_{2,3} = S^i_{1,3}$;

Round 3 : $S^i_{3,2}$ = Rotl_xxx_yy($S^i_{2,2}$ xor ($S^i_{2,3}$ and $S^i_{2,4}$) xor $S^i_{2,0}$ xor m_i, b_2);
$S^i_{3,0} = S^i_{2,0} <<< w_2$;
$S^i_{3,1} = S^i_{2,1}$;
$S^i_{3,3} = S^i_{2,3}$;
$S^i_{3,4} = S^i_{2,4}$;

Round 4 : $S^i_{4,3}$ = Rotl_xxx_yy($S^i_{3,3}$ xor ($S^i_{3,4}$ and $S^i_{3,0}$) xor $S^i_{3,1}$ xor m_i, b_3);
$S^i_{4,1} = S^i_{3,1} <<< w_3$;
$S^i_{4,0} = S^i_{3,0}$;
$S^i_{4,2} = S^i_{3,2}$;
$S^i_{4,4} = S^i_{3,4}$;

Round 5 : $S^{i+1}_{0,4}$ = Rotl_xxx_yy($S^i_{4,4}$ xor ($S^i_{4,0}$ and $S^i_{4,1}$) xor $S^i_{4,2}$ xor m_i, b_4);
$S^{i+1}_{0,2} = S^i_{4,2} <<< w_4$;
$S^{i+1}_{0,0} = S^i_{4,0}$;
$S^{i+1}_{0,1} = S^i_{4,1}$;
$S^{i+1}_{0,3} = S^i_{4,3}$;

VHDL code, testbench results & ATHENa reports:

Included on CD

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Evaluation of CAESAR candidates on FPGA

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2016**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Gorissen, Jasper

Datum: **15/01/2016**