An Optimized Hardware Implementation of Grain-128AEAD

Mattias Sönnerup
elt13ms1@student.lu.se
Ripudaman Khattar
ri0086kh-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors:
Martin Hell
Jonathan Sönnerup

Examiner:
Thomas Johansson

July 28, 2019

Printed in Sweden
E-huset, Lund, 2019

Abstract

With the Internet of Things era being here, more devices than ever are connected to the Internet, making the need for higher security greater. One important aspect of security to consider is encryption, which protects data from being read by unauthorized parties. Integrating cryptographic algorithms can be done in many ways and is usually dependent on restrictions of the area, speed, and power when it comes to hardware. One way of integrating security into hardware is with the use of stream ciphers, which can be very efficient in terms of power, area, and speed.

In this thesis, we present different implementations of Grain-128AEAD, which is a new stream cipher that can encrypt and decrypt data as well as provide authentication for the data. The implementations are optimized both on a hardware level, synthesis level and transistor level for different restrictions of area, power, and speed. The optimizations on hardware level include Galois transformation, isolation, pipelining and a transformation of the output function. For the synthesis level optimizations, we developed three scripts: one for reducing area/power with low power transistors (HVT) and two for increasing speed with high-speed transistors (LVT). On a transistor level, we optimized the Boolean expressions present in the feedback and output functions of the cipher.

In addition to that, we also present a new 64 times parallelization of Grain-128AEAD by allowing connections between the feedback and output functions of the cipher.

The combined effect of all the optimizations enabled the cipher to run at a throughput of 1.25-33.6 Gb/s compared to the 1-19 Gb/s throughput for the non-optimized versions. The area also improved by approximately 2-7% with the low area/power script, while the transistor level optimizations further reduced it by approximately 1-12% for the different parallelized versions. In addition to that, the power improved by up to 37% and 92% at 10 MHz and 100 KHz respectively.

Table of Contents

1 Introduction **1**
1.1 Purpose and Goals . . . 1
1.2 Thesis Outline . . . 2

2 Background **3**
2.1 Stream Ciphers . . . 3
2.1.1 Linear- and Nonlinear-Feedback Shift Register 3
2.1.2 Fibonacci Configuration 4
2.1.3 Galois Configuration 4
2.1.4 Output Function 4
2.1.5 Strength of a Cipher 4
2.2 Grain-128a . . . 5
2.2.1 Key and IV Loading 5
2.2.2 Feedback and Output Functions 5
2.2.3 Parallelization 6
2.2.4 Authentication 7
2.2.5 Keystream Generation 9
2.3 Grain-128AEAD . . . 9
2.3.1 Additional LFSR Initialization 9
2.3.2 Authentication 10
2.3.3 Keystream Generation 10
2.4 Additional Hardware . . . 10
2.4.1 Storage Elements 10
2.4.2 Multiplexer and Demultiplexer 10
2.4.3 State Machine 11
2.4.4 Counters 12
2.4.5 Clock Dividers 12
2.5 Design Properties . . . 13
2.5.1 Area 13
2.5.2 Speed 14
2.5.3 Power 14
2.6 Optimization Techniques . . . 15
2.6.1 Pipelining 15
2.6.2 Galois Transformation 16

2.6.3 Clock Gating 16

3 Implementation 19
3.1 Grain-128AEAD Implementation 19
3.1.1 Key and IV Loading 20
3.1.2 Initialization 20
3.1.3 Accumulator Loading 20
3.1.4 Keystream and MAC Generation 21
3.1.5 Encryption and Decryption 21
3.1.6 Tag Generation 22
3.2 Critical Path Analysis 22
3.2.1 Galois Transformation 22
3.2.2 Isolation of Authentication Section 25
3.2.3 Transforming and Pipelining Y 25
3.3 Modifying the Controller 26
3.4 Unrolling 28

4 Synthesis 29
4.1 Cipher Synthesis 29
4.1.1 Conditions and Restrictions 29
4.1.2 Compiling or Mapping the Design 30
4.1.3 Reporting the Results 30
4.2 Synthesis Scripts 31

5 Transistor Optimizations 33
5.1 MOSFET 33
5.1.1 Region of Operation of MOS Transistor 33
5.1.2 Techniques to Reduce Leakage Current 33
5.1.3 Techniques for Optimization Cipher Expressions 35
5.2 Boolean Expression Optimization for Grain-128AEAD 37

6 Result & Conclusion 41
6.1 Straightforward Implementation 41
6.1.1 High Speed 41
6.1.2 Low Power and Area 42
6.2 RTL Optimizations 44
6.2.1 High Speed 44
6.2.2 Low Power and Area 45
6.3 Transistor Level Optimizations 46
6.4 Analysis 47
6.5 Conclusion 48

References 51

A Code 55

B Transistor Optimization of the Grain-128AEAD Boolean Expressions 57

List of Figures

2.1 A Fibonacci LFSR. 4
2.2 A Galois LFSR. 4
2.3 An overview of the Grain-128a non-parallelized version [1]. 6
2.4 The Grain-128a parallelized 2 times [1]. 8
2.5 An overview of the authentication hardware [1]. 9
2.6 A figure of a multiplexer. 11
2.7 A figure of a demultiplexer. 11
2.8 A concept picture of a state machine. 12
2.9 A figure of a clock divider by 2. 13
2.10 A figure of a 4 bit power of 2 clock divider using a counter. 13
2.11 Leakage types in the MOSFET. 15
2.12 Clock gating using an OR gate (left) and an AND gate (right) . . 17

3.1 An overview of the non-parallelized Grain-128AEAD implementation without any optimization. 20
3.2 n and $n/2$ shift selection hardware. 21
3.3 A figure showing the possible paths for optimization of the Grain-128AEAD implementation. 23
3.4 An overview of the isolation of the authentication state for the $n > 2$ parallelized versions. 25
3.5 An overview of y transformation and pipelining [1]. 26
3.6 An overview of the modified controller. 27
3.7 A figure showing the concept of unrolling for the g function at $n = 33$. 28

4.1 Synopsis DC flow diagram. 30
4.2 Flow diagram for optimizing Grain128-AEAD. 31

5.1 Transistor region of operation. 34
5.2 De Morgan's theorem. 35
5.3 Standard gate design using pass transistor. 36
5.4 Level restorer. 36
5.5 Transmission Gate circuit diagram and symbol. 37
5.6 Boolean expression to transistor level. 37

5.7 A figure of a boolean expression “AB xor ACD” optimized at transistor level. 38
5.8 A figure of a boolean expression “AB xor CD xor EF xor GH” optimized at transistor level. 39

B.1 A figure of a boolean expression “ABCD xor EFG” optimized at transistor level. 57
B.2 A figure of a boolean expression “AB xor CD” optimized at transistor level. 58
B.3 A figure of a boolean expression “ABC xor DE” optimized at transistor level. 59
B.4 A figure of a boolean expression “AB xor CD xor EF” optimized at transistor level. 59

List of Tables

5.1 Optimization technique for leakage current. 34

6.1 High speed result for the straightforward implementation. 42
6.2 High speed result for the straightforward implementation using the modified controller. 42
6.3 Result for the straightforward implementation running at 100 kHz with the speed script compared to the power/area script using the standard and modified controller. 43
6.4 Result for the straightforward implementation running at 10 MHz with the speed script compared to the power/area script using the standard and modified controller. 43
6.5 High speed result for the optimized implementation. 44
6.6 High speed result for the optimized implementation using the modified controller. 45
6.7 Result for the optimized implementation running at 100 kHz with the power/area script for the standard controller and the modified controller compared to the best result from the straightforward implementation. 45
6.8 Result for the optimized implementation running at 10 MHz with the power/area script for the standard controller and the modified controller compared to the best result from the straightforward implementation. 46
6.9 Transistor count and gate count for the ST65nm technology. . . . 47
6.10 Gate count for Grain-128AEAD without transistor level optimization compared to with transistor level optimization. 47

Chapter 1
Introduction

In an industry where technology is becoming more advanced, security is starting to become more of an issue. With the Internet of Things (IoT) era being here more devices than ever are being connected to the Internet, making the need for security greater. Companies are constantly looking for the best way to implement security based on their own restrictions of chip cost/area, speed, and energy efficiency.

Security can be achieved and defined in many ways, but one important aspect of security is encryption. Encryption protects data from being read by unauthorized parties. Encryption can be done in many stages of the design for a device, but the earlier you do it the more secure and faster your device is. A simple and fast way to encrypt data is using stream ciphers.

When companies and organizations want to make stream ciphers to be used in their design, they usually have to implement and test out a few different stream ciphers to see which one works best on their own restrictions of speed, area or energy efficiency. Implementing and trying out many versions of different stream ciphers can be very time consuming.

Our contribution with this thesis is to aid in this work by implementing a new variant of the Grain-128a stream cipher called Grain-128AEAD and optimizing it as much as possible based on different restrictions on area, power and speed. The optimization will be made on not only a gate level but will also be on a transistor level, which is something that can often be neglected when it comes to optimizing stream ciphers. The idea is that the result and process presented in this thesis can be used to compare with other stream ciphers implemented under similar restrictions.

1.1 Purpose and Goals

The main goals of the thesis can be summarized as follows:

- Optimize Grain-128AEAD for speed, area and power through VHDL and synthesis.
- Analyze and optimize the hardware implementation of the Boolean functions used in Grain-128AEAD on a transistor level.
- Implement different synthesis scripts for high speed, low area and power.

- Make a comparision between the implementations to determine the best implementation based on power, area, speed and script.

1.2 Thesis Outline

Chapter 2 (Background) provides the cryptography and hardware background needed to understand the thesis.

Chapter 3 (Implementation) presents the implementation as well as optimizations that can be performed on Grain-128AEAD.

Chapter 4 (Synthesis) describes the synthesis of the Grain-128AEAD implementation and the different synthesis scripts that are used to obtain the result on restrictions of speed as well as restrictions on area and power.

Chapter 5 (Transistor Level Optimizations) shows the general process of how to find transistor level optimizations.

Chapter 6 (Results and Conclusion) presents the result as well as some discussions regarding the result obtained from synthesis and the transistor level optimizations.

Chapter 2
Background

To get a clear and full picture of what the thesis is about, it is essential to explain some general concepts and definitions in the subject of cryptography as well as describe some basic hardware implementations. In this chapter stream ciphers are first described in a general sense. After that details about the design of the Grain-128AEAD stream cipher are provided by going over the design of Grain-128a and then describing the differences to Grain-128AEAD. Next, general theory is provided on some standard hardware components and methods that will be used in the implementation. Finally, some theory on area, speed, and power together with a few optimization techniques used in the implementation section is described.

2.1 Stream Ciphers

A stream cipher [2] is a class of encryption algorithms that generates a time varying pseudo-random stream of digits (keystream) and use it for encryption/decryption of messages. Encryption of messages ensures that it cannot be read by any unauthorized parties. A plaintext message, m_i can be encrypted in a stream cipher by XORing each bit in the message, bit-wise, with each bit of the keystream, y_i, as

$$c_i = m_i \oplus y_i, \tag{2.1}$$

where c_i is the encrypted text. Consequently, a ciphertext can be decrypted as

$$m_i = c_i \oplus y_i. \tag{2.2}$$

2.1.1 Linear- and Nonlinear-Feedback Shift Register

Stream ciphers can be constructed in multiple ways. Some common ways of construction a stream cipher includes using Linear-Feedback Shift Registers (LFSRs), Nonlinear-Feedback Shift Registers (NFSRs) or a combination of both. An LFSR [3] and NFSR [4] is a shift register that has an input which is a function of its own state, often referred to as a feedback function. The difference between an LFSR and NFSR is that an LFSR only consists of linear gates like XORs in its feedback function, while an NFSR can have nonlinear gates like AND gates and Inverter gates as well. The bits of the shift registers that influence the inputs are called taps. The selection of the taps together with the configuration of the

NFSR/LFSR will determine the next state of the shift registers which can then be used by an output function to generate the pseudo-random data.

2.1.2 Fibonacci Configuration

There are two possible configurations of LFSRs and NFSRs. One configuration is the Fibonacci configuration [5], as seen in Figure 2.1. In the Fibonacci configuration, the bit on the far-right side is called the output bit. The output bit is fed back through a gate, together with each of the taps, into the bit position on the far-left side which is called the input. Due to the fact that the output bit has to go through a sequence of gates with each tap just to get to the input, the critical path will increase for every tap in the shift register.

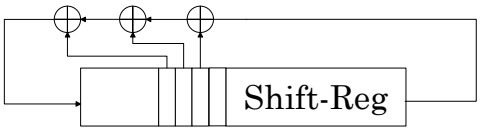


Figure 2.1: A Fibonacci LFSR.

2.1.3 Galois Configuration

Another configuration is Galois, which can be seen in Figure 2.2. In the Galois configuration [6], there are instead multiple feedback functions between each tap, which allows for a smaller critical path. Galois can however not be parallelized as many times as the Fibonacci configuration.

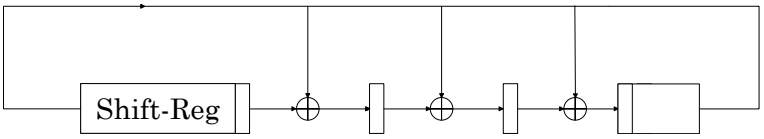


Figure 2.2: A Galois LFSR.

2.1.4 Output Function

The output function consist of a series of XOR, AND, OR and/or NOT gates. It is used to generate the keystream bits for the encryption/decryption of messages. Usually, the keystream bits can be generated after the IV and key has been loaded, but for certain ciphers some additional initialization rounds are required before the output function bits are considered to be keystream bits.

2.1.5 Strength of a Cipher

The strength of a keystream and the stream cipher itself is determined by how random the sequence of ones and zeros generated from a keystream are. Nonrandomness in the keystream can often be exploited in order to mount attacks on the

cipher. Since the keystream is directly involved in the encryption and decryption of data, it has to be as random as possible and be kept a secret for outsiders who are not authorized to read it. Unfortunately, a purely random keystream can not be generated by stream ciphers. Stream ciphers by definition only generate a pseudo-random keystream as an output. What that means is that the output of the stream cipher appears to be random for an outsider, but is really just generating the same output for the same key and IV. The downside of a pseudo-random process is therefore that if an unauthorized person gets a hold of the key and IV sent into a stream cipher, they can generate the same keystream and use it to decrypt any data that has been encrypted by the keystream in the past.

2.2 Grain-128a

Grain-128a [1] is a new version of the Grain-128 [7] stream cipher with two modes: with authentication or without authentication. It consists of two feedback registers, an NFSR and an LFSR that are both 128 bits long, and an output function. The content of the NFSR is denoted as $b_i, b_{i+1}, ..., b_{i+127}$, and the content of the LFSR is denoted as $s_i, s_{i+1}, ..., s_{i+127}$.

2.2.1 Key and IV Loading

The feedback registers are loaded by two values: a 128 bit `key` denoted as k_i, $0 \leq i \leq 127$, and a 96-bit initialization vector (`IV`) denoted as IV_i, $0 \leq i \leq 95$. The loading of the key and IV bits are performed as follows. The 128 k_i bits are sent into the NFSR as $b_i = k_i$, $0 \leq i \leq 127$, and the 96 IV_i bits are sent into the first 96 bits of the LFSR as $s_i = IV_i$, $0 \leq i \leq 95$. The last 32 bits of the LFSR are set to $s_i = 1$ for $96 \leq i \leq 126$ and $s_{127} = 0$ for the last bit.

2.2.2 Feedback and Output Functions

The feedback function for the LFSR and NFSR is defined in Eq. 2.3 and Eq. 2.4. Moreover, Grain-128a consists of one output function that generates the output for the cipher. The corresponding output function for Grain-128a can be implemented using Eq. 2.5.

In both the feedback functions, the y_{out} signal, which is the result from the output function, is fed back as y_{feedback} during the `initialization` state of the stream cipher for the first 256 bits. Otherwise, the y_{feedback} signal is not fed back to the shift registers.

In Figure 2.3 a picture of Grain-128a together with its feedback and output functions can be seen.

Note that all expressions above assumes that the most significant bit, s_{i+127} and b_{i+127}, is to the right.

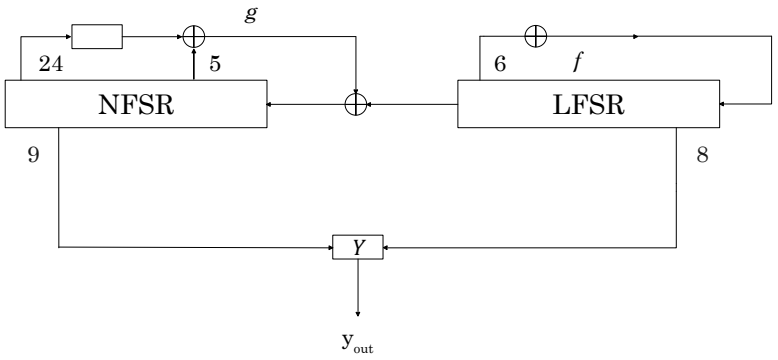


Figure 2.3: An overview of the Grain-128a non-parallelized version [1].

$$
\begin{aligned}
f_{\text{out}} = {} & s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96} \\
& + y_{\text{feedback}},
\end{aligned} \tag{2.3}
$$

$$
\begin{aligned}
g_{\text{out}} = {} & s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} \\
& + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} \\
& + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} \\
& + b_{i+68}b_{i+84} + b_{i+88}b_{i+92}b_{i+93}b_{i+95} \\
& + b_{i+70}b_{i+78}b_{i+82} + b_{i+22}b_{i+24}b_{i+25} \\
& + y_{\text{feedback}}.
\end{aligned} \tag{2.4}
$$

$$
\begin{aligned}
y_{\text{out}} = {} & b_{i+12}s_{i+8} + s_{i+13} + s_{i+20} + b_{i+95}s_{i+42} \\
& + s_{i+60}s_{i+79} + b_{i+12} + b_{i+95}s_{i+94} + s_{i+93} \\
& + b_{i+2} + b_{i+15} + b_{i+36} + b_{i+45} + b_{i+64} + b_{i+73} \\
& + b_{i+89},
\end{aligned} \tag{2.5}
$$

$$
y_{\text{feedback}} = \begin{cases} y_{\text{out}}, & \text{for the first 256 bits.} \\ 0, & \text{otherwise.} \end{cases} \tag{2.6}
$$

2.2.3 Parallelization

Grain-128a can also be implemented with a throughput of n output bits, where $n = \{1, 2, 4, 8, 16, 32\}$. Implementations, where n is higher than 1, is done through parallelization. Figure 2.4 shows how a two-times parallelized Grain-128a is implemented. As can be seen by the figure the two feedback functions and the output function are offset to the left by 1, each leaving behind another feedback/output function in its place.

For n times parallelization, the concept is very similar. The two feedback functions and output function needs to be offset $n-1$ times to the left, leaving behind $n-1$ functions each. As a result of this, as throughput increases so does the amount of hardware and the area. Apart from the changes to the design the NFSR and LFSR have to be modified to shift n bits to the left instead of just shifting one bit at a time.

The expressions for the LFSR-feedback, NFSR-feedback and output functions can be described as in Eq. 2.7, Eq. 2.8 and Eq. 2.9, respectively.

$$\begin{aligned} f_{\text{out}}^k = {} & s_{i+k} + s_{i+7+k} + s_{i+38+k} + s_{i+70+k} + s_{i+81+k} + s_{i+96+k} \\ & + y_{\text{feedback}}. \end{aligned} \tag{2.7}$$

$$\begin{aligned} g_{\text{out}}^k = {} & s_{i+k} + b_{i+k} + b_{i+26+k} + b_{i+56+k} + b_{i+91+k} + b_{i+96+k} \\ & + b_{i+3+k}b_{i+67+k} + b_{i+11+k}b_{i+13+k} + b_{i+17+k}b_{i+18+k} \\ & + b_{i+27+k}b_{i+59+k} + b_{i+40+k}b_{i+48+k} + b_{i+61+k}b_{i+65+k} \\ & + b_{i+68+k}b_{i+84+k} + b_{i+88+k}b_{i+92+k}b_{i+93+k}b_{i+95+k} \\ & + b_{i+70+k}b_{i+78+k}b_{i+82+k} + b_{i+22+k}b_{i+24+k}b_{i+25+k} \\ & + y_{\text{feedback}}. \end{aligned} \tag{2.8}$$

$$\begin{aligned} y_{\text{out}}^k = {} & b_{i+12+k}s_{i+8+k} + s_{i+13+k} + s_{i+20+k} + b_{i+95+k}s_{i+42+k} \\ & + s_{i+60+k}s_{i+79+k} + b_{i+12+k} + b_{i+95+k}s_{i+94+k} + s_{i+93+k} \\ & + b_{i+2+k} + b_{i+15+k} + b_{i+36+k} + b_{i+45+k} + b_{i+64+k} + b_{i+73+k} \\ & + b_{i+89+k}. \end{aligned} \tag{2.9}$$

$$y_{\text{feedback}}^k = \begin{cases} y_{\text{out}}^k, & \text{for the first 256 bits .} \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

where $0 \leq k \leq 31$.

Note that it is once again assumed that the most significant bit, s_{i+127} and b_{i+127}, are to the right in Figure 2.3.

Parallelization beyond 32 is possible but requires a different approach. The approach is described in Section 3.4.

2.2.4 Authentication

Authentication can be performed after the 256 bits have been fed back to the feedback functions in the `initialization` state, but only when $IV_0 = 1$. When $IV_0 = 0$, authentication cannot be performed and is forbidden according to its design details. During authentication 64 output bits from the Grain-128a output function is taken and inputted into two registers: an accumulator and a shift register, which are both 32 bits. The first 32 bits go into the accumulator as

$$a_0^j = y_{256+j}, \tag{2.11}$$

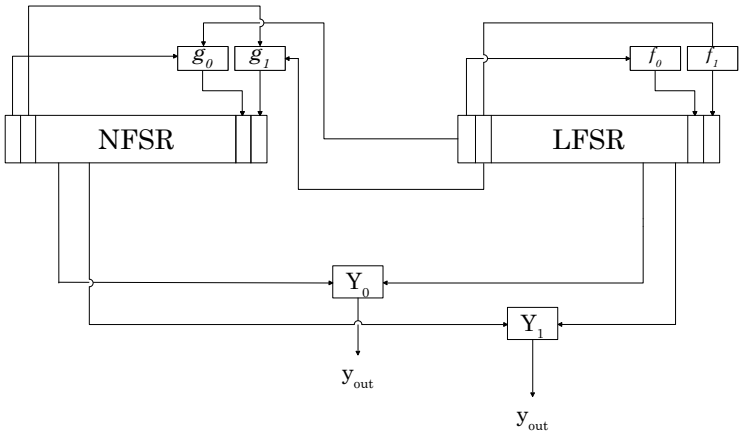


Figure 2.4: The Grain-128a parallelized 2 times [1].

where $0 \leq j \leq 31$, denotes the index of accumulator register. Consequently, the last 32 bits go into the shift register as

$$r_i = y_{288+i}, \tag{2.12}$$

where $0 \leq i \leq 31$, denotes the index of shift register. Note that bit y_{256} is the first bit after initialization and bit y_{288} is the first bit after loading the accumulator.

After the initialization of the registers, a message of length L defined as m_i, $0 \leq i \leq L-1$ with a high bit padded to its end, $m_L = 1$, is sent into the authentication hardware. The message, as well as the shift registers, are used to update the accumulator register as

$$a_{i+1}^j = a_i^j + m_i r_{i+j}, \tag{2.13}$$

where $0 \leq i \leq L$, $0 \leq j \leq 31$, j denotes the index of accumulator register and r_{i+j} denotes the content of the shift register at index j.

The shift register is updated by shifting in every second bit after its been loaded, $y_{320+2i+1}$.

It is also possible to parallelize the authentication hardware up to 16 times with a 32-bit parallelized Grain-128a since the authentication hardware only accumulates with every second bit. The expression for n times authentication parallelization is

$$a_{i+1}^j = a_i^j + \sum_{k=0}^{n-2} m_{i+k} r_{i+j+k}. \tag{2.14}$$

Note that $n-1$ additional bits are required, $r_{i+32}, ..., r_{i+32+n-2}$, which refers to the first $n-1$ values that are going to be shifted into shift register.

After all message bits have been sent in, the final content of the accumulator register, $a_{L+1}^0, ..., a_{L+1}^{31}$, is denoted as the tag. The tag will then be used for authentication.

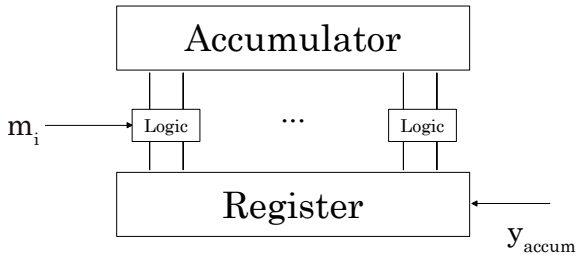


Figure 2.5: An overview of the authentication hardware [1].

2.2.5 Keystream Generation

For Grain-128a the keystream or the output from the design, used for encryption, is defined differently depending on whether or not authentication is used. When authentication is not used, the keystream of Grain-128a is defined as

$$z_i = y_{256+i} \tag{2.15}$$

However, when authentication is used, the keystream of the Grain-128a is defined as

$$z_i = y_{320+2i}, \tag{2.16}$$

which means that only every second bit after the 64 bits used to initialize the accumulator and shift registers, are used.

2.3 Grain-128AEAD

Grain-128AEAD is a newer version of Grain-128a. The design of the stream cipher is the same as depicted in Figure 2.3. It contains an LFSR, NFSR, two feedback functions and one output function that are identical to the functions used in Grain-128a. There are mainly three differences between both designs: An additional initialization of the LFSR; The increase from a 32-bit to a 64-bit accumulator and shift register; A change of how the keystream is generated.

2.3.1 Additional LFSR Initialization

The additional LFSR initialization [8] is performed during the loading of the accumulator and shift register. The goal behind it is to prevent attacks such as state recovery attacks from leaking the key. For this cipher, the key is XORed into the LFSR state through its feedback functions as

$$s_i^* = s_i + k_i, \tag{2.17}$$

where s_i^* is the new value of the LFSR state at index i after the initialization.

2.3.2 Authentication

The authentication process of Grain-128AEAD closely resembles the process for Grain-128a in Section 2.2.4. The accumulator is initialized as in Eq. 2.11 with $0 \leq j \leq 63$. Consequently, the shift register is initialized as in Eq. 2.12 with $0 \leq i \leq 63$. Following the initialization of the registers, the accumulator is updated in the same fashion as in Eq. 2.13, with $0 \leq j \leq 63$. Similarly, the shift register is updated by shifting in every second bit, $y_{384+2i+1}$.

2.3.3 Keystream Generation

For Grain-128AEAD, the keystream is generated as

$$z_i = y_{384+2i}, \tag{2.18}$$

when using and not using authentication.

2.4 Additional Hardware

Apart from the hardware belonging to the stream cipher, there are several other types of hardware that can or needs to be used in the design of most hardware implementations. For the ciphers, additional hardware will be required for loading, initialization and keystream generation. In this section, all hardware used in our implementation of the stream cipher is briefly described so that there is no ambiguity surrounding the implementation of the ciphers in Chapter 3.

2.4.1 Storage Elements

In digital electronics there are two types of memory elements: flip-flops [9] and latches [10]. Latches are the basic storage element that operates with a level sensitive enable signal. They are commonly used in the design of asynchronous sequential circuits. Moreover, latches are usually faster in comparison to flip-flops because they don't need to wait for a clock signal, due to this property they are generally preferred in high-speed designs. In low area and speed designs, latches are also preferred with their small die-size and low power consumption. The last advantage of a latch is the Time-Borrowing property, in which if an operation is not completed within a certain time, the required time for executing the operation is borrowed from the other operational time.

Flip-flops are binary storage devices which can store a high or a low value. It differs from a latch in that it has a clock input and is edge triggered. Being an edge triggered storage device means that when the clock goes from high to low or low to high its value updates, which makes it a clock-controlled memory device.

2.4.2 Multiplexer and Demultiplexer

A multiplexer [11] or a mux is a digital component that takes a number of input signals and forwards one of the inputs to the output signal. The input that is

selected as the output signal is determined by an additional signal the select signal. A figure of a multiplexer can be seen in Figure 2.6.

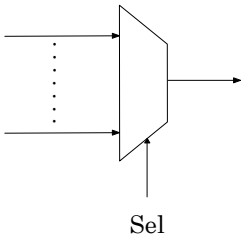


Figure 2.6: A figure of a multiplexer.

The opposite of a multiplexer is a demultiplexer [11]. A demultiplexer takes one input signal and forwards it to a different output signal depending on the select signal. A figure of a demultiplexer can be seen in Figure 2.7.

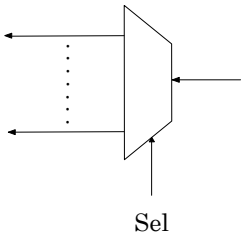


Figure 2.7: A figure of a demultiplexer.

Multiplexers and demultiplexer are both very commonly used in designs since they allow for the re-routing of signals.

2.4.3 State Machine

There are multiple ways to control a design. One way is using combinational logic which takes an input signal and sends out the desired output signal, which can be routed as a control signal for components like multiplexers. However, for certain designs controlling a circuit based solely on the current inputs is simply not enough. For those designs, an alternative solution is using sequential control logic that takes not only an input but also a sequence of previous inputs in order to determine the output signal. When it comes to sequential control logic, state machines [12] are the most widely used, often forming the core of a magnitude of digital systems today.

A state machine [13] consists of two main blocks: a memory, often registers, and a combinational block. The memory block is used to store values that correspond to the state of the design. The combinational block can be seen as two different blocks. One block is used to determine the next state of the design, based on the current state as well as the input to the state machine, while the other block is used to generate the output of the machine. In Figure 2.8 a concept picture is presented of how a state machine is built. The actual hardware used for the state

machine depends very much on the use case. Some state machines are simple and others are complex depending on how much of the design needs to be controlled.

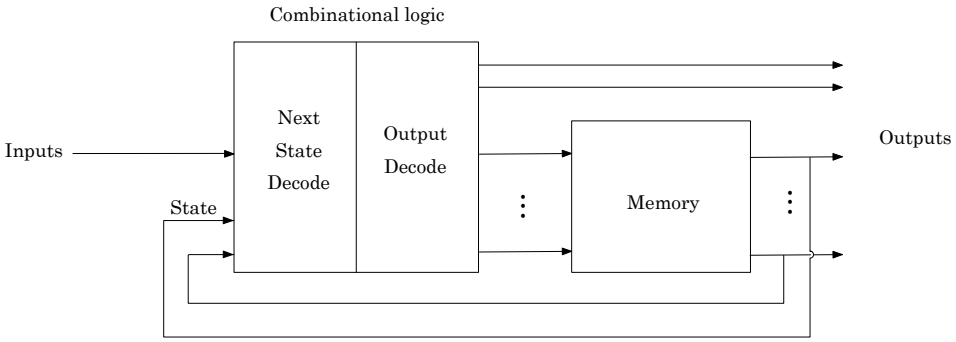


Figure 2.8: A concept picture of a state machine.

2.4.4 Counters

A counter is a sequential circuit which is used for counting events like for example clock cycles. They are designed using flip-flops with a clock signal applied. Counters are of two types asynchronous or ripple counters, and synchronous counters. In an asynchronous counter, there is no universal clock, only the first flip-flop is given a clock pulse, while the other flip-flops are dependent upon the output of previous flip-flops. The flip-flop which is applied with the main clock pulse acting as an LSB (least significant bit) in the counting sequence.

In the synchronous counter, the universal clock signal is given to all the flip-flops so that the output changes in parallel. This makes a synchronous counter operate at a higher speed in comparison with an asynchronous counter. These types of counters are easy to design and less prone to the race condition, which makes them more reliable compared to asynchronous counters.

2.4.5 Clock Dividers

A clock divider [14], also known as a frequency divider, is a circuit used to divide a clock signal with frequency, f, into a new clock signal with frequency f/k, where k is an integer. It can be used in many circuits where a design needs to be slowed down or where an additional, lower frequency, clock is required. There are different types of clock dividers with different divisions like division by two and division by the power of two.

Clock division by two is a simple divider made by a flip-flop and an inverter. When a high clock signal is received, the content of the flip-flop is inverted and sent out. The value sent out will be a pulse that is high for one period and low for another period of the input clock signal, making the pulse a clock signal with two times the period and two times less frequency. A figure of the clock divider with a division by two can be seen in Figure 2.9.

Clock division by a power of two is a little more complex. It requires an n-bit counter to achieve a clock division by 2^n. The benefit of a power of two counter

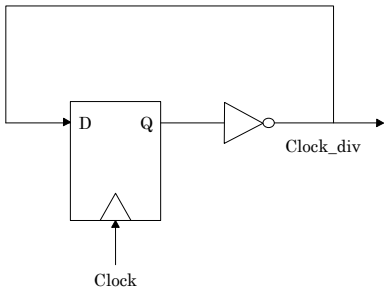


Figure 2.9: A figure of a clock divider by 2.

is that it can be used to generate all clock divisions by a factor of 2^k, where $1 \leq k \leq n$, without the use of any extra hardware. By looking at the content of the counter it can be seen that each bit changes values every $(2^k)/2$ clock cycles, making the content of the counter bit at position k a perfect clock divider signal by 2^k. A figure of a power of two clock divider can be seen in Figure 2.10.

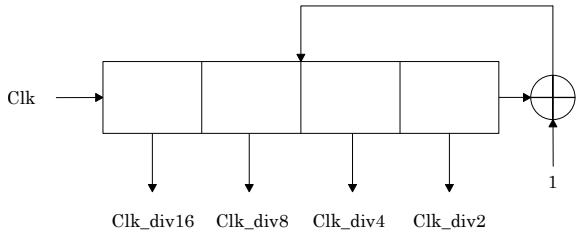


Figure 2.10: A figure of a 4 bit power of 2 clock divider using a counter.

2.5 Design Properties

In the design there are multiple properties that can be extracted from an implementation. Some properties include area, speed, and power consumption. In this section, a short overview of these three properties is presented as well as some general theory on how to optimize them.

2.5.1 Area

The area [15] of a design depends on multiple things. In general, the area is defined as the space on a circuit or chip that the hardware takes up. How much space the hardware take up is dependent on a lot of things. For ASIC the area is dependent on the transistor technology and size that is used to make up the logical gates of a design. Whereas, for FPGA (Field Programmable Gate Array) the area is dependent on the number of resources the design use inside the FPGA. For stream ciphers, it is difficult to reduce the area used in the design of the cipher since registers and all logic functions in the design are of importance for the security

of the cipher. Changing any part of the general design can alter the security of the cipher, which needs to be taken into consideration when optimizing for area. It is, however, possible to improve the area of an implementation by making sure all other logic that is required for implementing the ciphers in hardware takes up as little space as possible. The best way to do it is to make sure as much hardware as possible is being shared. An alternative is also to reduce the number of transistors in the gates that make up the hardware. For the stream cipher, the amount of hardware cannot be reduced in the design and there is not much hardware that can be shared. However, some transistor-level optimizations on the Boolean expressions of the functions are later presented in Chapter 5.

2.5.2 Speed

Speed [16] is determined by the critical path, which is the longest path between an input to a register, register to output or input to output, of a design. Reducing the critical path of a design can allow it to function for a much higher clock frequency and thus become faster. For certain designs like stream ciphers speed is often defined by how many bits of data is sent out per time unit, which is called the throughput of the design. In the case of stream ciphers, the throughput is defined as the number of keystream bit sent out every second.

2.5.3 Power

Power [17] exist in many forms. In CMOS circuits, power is dissipated in two forms: dynamic and static power.

$$P_{\text{total}} = P_{\text{dynamic}} + P_{\text{static}}. \tag{2.19}$$

The dynamic power is dissipated when transistors change their state as

$$P_{\text{dynamic}} = P_{\text{switching}} + P_{\text{short-circuit}}. \tag{2.20}$$

The switching power is the power dissipated due to charging and discharging of capacitance present inside an integrated circuit. An expression for switching power is given as

$$P_{\text{switching}} = \alpha f C V_{dd}^2, \tag{2.21}$$

where α is the switching activity, f is the switching frequency, C is the overall capacitance and V_{dd} is the supply voltage.

The power is dissipated by the occurring of a short circuit connection between the main voltage and the ground when the gate changes its state is known as short-circuit power and is given as

$$P_{\text{short-circuit}} = I_{sc} V f, \tag{2.22}$$

where I_{sc} is the short-circuit current, V is the power supply and f the switching frequency.

The static power occurs when a circuit is in steady state and the circuit is turned on. Static power occurs because of leakage current [18] in the MOSFET. The leakage current depends upon different factors like change in threshold voltage,

supply voltage, the thickness of the oxide layer and the doping profile as mentioned in Figure 2.11. An expression for static power is given in as

$$P_{\text{static}} = (I_{\text{sub}} + I_{\text{junction}} + I_{\text{gate}})V_{DD}. \tag{2.23}$$

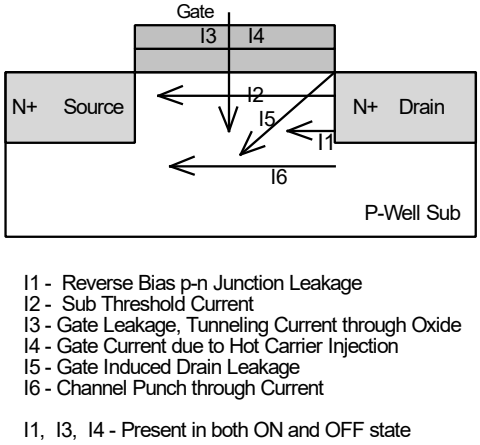


Figure 2.11: Leakage types in the MOSFET.

2.6 Optimization Techniques

When making a hardware implementation it is beneficial to know about some of the optimization techniques which can be applied to the implementation in order to improve the performance of the system. In the industry integrated circuits has certain restrictions on the area, speed, and power and it depends on what context the electronics hardware is used for. In the case of charging devices, it might not be too hard to assume that low power is preferred so that the energy of the electronics devices last longer. For servers or networking devices that need to transmit a lot of data to multiple clients, power might not be the main concern but rather the speed of the system without losing the data. In the following subsections, a few optimization techniques are presented which are useful in optimizing the stream cipher implementation in Chapter 3.

2.6.1 Pipelining

Pipelining [19] is a technique used to increase speed and reduce power by strategically inserting additional flip-flops at a certain part of a design. Adding additional flip-flops along combinational paths, that contribute to the critical path, can significantly improve the speed/throughput of the design by shortening the propagation delay. Decreasing the critical path can lead to a decrease in power as well since it can allow for a much lower supply voltage for a circuit. Inserting additional

flip-flops can also be beneficial to reduce glitches. Glitches [20] are present in every implementation and are caused by converging combinatorial paths that have different propagation delays, which causes unwanted transitions in gates that consumes power. Pipelining can interrupt the propagation of glitches and thus reduce the power consumption caused by them. Unfortunately, pipelining a design is not always possible since it adds an extra clock cycle of delay for the signals. For the stream ciphers, pipelining can not be done on feedback functions without altering the functionality of the cipher. It can, however, be applied to other parts of the design where adding an extra delay to the signal does not affect the functionality of the cipher.

2.6.2 Galois Transformation

Galois transformation is a technique that can be used to transform Fibonacci configurations of a feedback-shift-register into its Galois equivalent. The transformation is performed by simply adjusting the initial state as well as reverse the order of all the connections to feedback function and then separate each feedback function into multiple new functions with different connections. By performing the transformation it can allow for a feedback shift register to run on a much higher frequency and possibly the stream cipher itself, which makes it a good technique to use when implementing fast stream ciphers. However, in order to select the new feedback functions certain conditions have to be met. To guarantee that a feedback shift register maintains its equivalence after and before shifting, no bits with a lower index than the terminal bit, τ, can be shifted to. The terminal bit is defined as

$$\tau = \max(\text{index}_{\max}(p) - \text{index}_{\min}(p)), \quad \forall p \in P, \tag{2.24}$$

where p is all the product-terms in the Fibonacci feedback function, $\text{index}_{\max}(p)$ and $\text{index}_{\min}(p)$ denotes the minimum and maximum index of a term in p. In addition to that, the transformation has to guarantee that all of the internal state bits of the feedback shift register, as well as the bits used for generating the output, is equivalent before and after the transformation. For Grain-128AEAD the terminal bit would then have to be selected as 96 since that is the last bit used by the output function. It is possible to extend this theory for parallelized versions, by restricting the transformation connection only to bits from where feedback is possible [21].

It is important to note that for LFSRs the Galois transformation is unique, however, for NFSR it is not unique. This allows for a multiple of transformations to be used in the NFSR case. An algorithm for finding the fastest NFSR transformation is found in [21] and an algorithm for the most speed and area efficient transformation can be found in [22].

2.6.3 Clock Gating

Clock gating [23] is a low-power technique that is used to reduce power consumption. The technique reduces the power consumption by disabling or suppressing

the clock for certain parts of the design when they are not used. This prevents unnecessary transitions from occurring in sequential elements like flip-flops, in those certain parts, because the clock is not active when they occur. Since all transitions that occur in sequential elements consumes power, power will be saved by preventing these transitions from occurring. Clock gating on a design is only possible if there are sequential elements within the design that are unused in certain clock periods. A basic implementation of clock gating using an AND-gate or an OR-Gate can be seen in Figure 2.12.

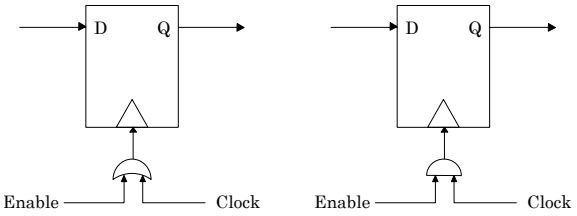


Figure 2.12: Clock gating using an OR gate (left) and an AND gate (right)

Chapter 3
Implementation

So far the general cryptography and hardware theory, together with the design details have been presented in order to get a general idea of how to make an implementation. However, as previously described in Chapter 2, ciphers require additional control hardware than that presented in the design details to regulate when and how it needs to be loaded, initialized and output the keystream. It is largely dependent upon the designer to make an efficient implementation that yields the best performance in terms of area, speed, and energy efficiency. In this chapter, we present implementations of the Grain-128AEAD cipher from Chapter 2, as well as the application of some hardware efficient optimizations, in order to improve the performance of the designs.

All implementations presented in this chapter are designed at Register Transfer Level (RTL) using VHDL and are verified by simulation in both Vivado v2016.1 and Modelsim 10.0d.

3.1 Grain-128AEAD Implementation

An implementation of Grain-128AEAD, without any optimization, can be seen in Figure 3.1. It consists of all the hardware presented in Figure 2.3 with the addition of some extra hardware used for controlling the cipher, mainly the controller. In the design, there are multiple muxes, regulated by the controller, which are used for managing the input to the registers in addition to the output of the design. There are many ways to implement a controller, but the most common practice is to use a state machine [13] and/or a counter [24]. To begin with, our Grain-128AEAD controller uses a state machine together with a counter to keep track of and decide when to switch state. Later on, an alternative controller is explored in Section 3.3.

The controller to begin with consists of five states: `reset`, `loading`, `initialization`, `accumloading` and `normal` state. The controller starts off in the `reset` state. In the `reset` state, the controller blocks off the muxes, making the input of the register as well as the output of the design zero. Once an enable signal is received from outside the design, the controller moves on to the `loading` state.

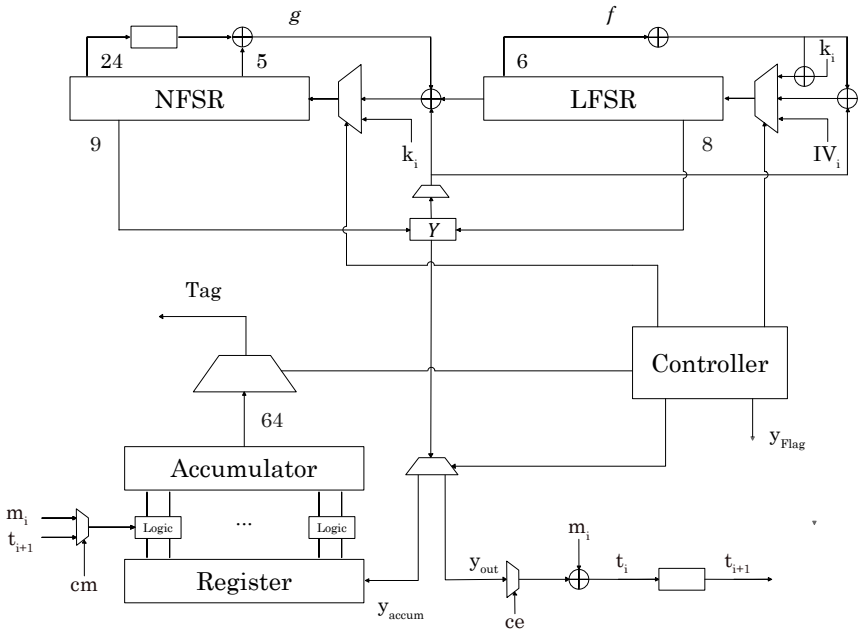


Figure 3.1: An overview of the non-parallelized Grain-128AEAD implementation without any optimization.

3.1.1 Key and IV Loading

During the `loading` state, the controller opens up the muxes connected to the NFSR and LFSR for the key and IV respectively. For the non-parallelized version, the key and IV is loaded one bit at a time per clock cycle. However, in the n-parallelized version, the key and IV is loaded by n bits per clock cycle instead of just one. The reason for that is because the number of loading bits per clock cycles for the parallelized versions does not have a significant effect on the area and also decreases the bottleneck for the overall design drastically. Finally, when loading is finished the design moves onto the `initialization` state.

3.1.2 Initialization

In the `initialization` state, the controller opens up the mux for the y value so it is XORed with the output of the f and g function, which is then sent into the LFSR and NFSR, respectively. After 256 clock cycles have passed or after 256 y bits have been generated for the parallelized versions, the controller moves on to the `accumloading` state.

3.1.3 Accumulator Loading

When in the `accumloading` state, the controller opens up the mux connected to the shift register for all the y values sent out by the design. The bits then get shifted into the shift register n bits at a time. When 64 bits have been loaded

into the shift register, the message m_i gets set to 1 by the controller. Setting the message to one at that specific time allows the content of the shift register to be loaded into the accumulator according to Eq. 2.13. Note that the accumulator content is zero at that point. After the loading of the accumulator, the controller stays in the `accumloading` state until 64 more values have been shifted into the shift register so it gets loaded as well. Throughout the `accumloading` process, the key XORed with the output from the f function is fed back to the LFSR. To save area, the key is sent in once again from the start of the `accumloading` state up until all key bits have been sent in, which happens to be exactly when the accumulator and shift register have finished loading. Moreover, a signal y_{flag} is also sent out at the start of the `accumloading` state to notify from outside the design that the key bits need to be sent in again. After the key bits have been sent in and the accumulator and shift register are loaded, the design moves on to the `normal` state.

3.1.4 Keystream and MAC Generation

In the `normal` state, all the y bits following the loading of the accumulator and shift register are divided up into two groups: keystream bits used for encryption/decryption and accumulator/MAC bits used for generating the tag. Every second y bit becomes a keystream bit and goes to the y_{out} signal, while every other bit becomes a MAC bit and goes to y_{accum}.

For the non-parallelized version, it can be noted that the accumulator and shift register needs to be slowed down by a factor 2 through a clock divider. The reason for the slowdown is to prevent the shift register from shifting when a y_{accum} value is not generated. For the parallelized versions, a clock divider is not used during authentication. Instead, the controller alters the number of shifts of the shift register to $n/2$ during the `normal` state, since only half of all bits go to the accumulator every clock cycle. The additional hardware needed for selecting between an n or an $n/2$ shift for the parallelized versions can be seen in Figure 3.2.

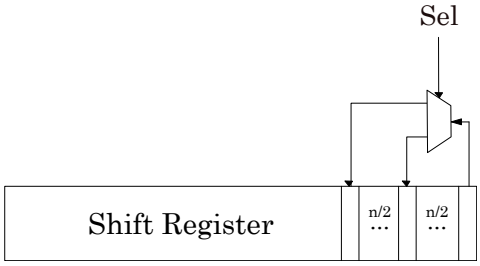


Figure 3.2: n and $n/2$ shift selection hardware.

3.1.5 Encryption and Decryption

Encryption and decryption can be performed after the generation of each y_{out} bit in the `normal` state. It is controlled by the signals *ce* and *cm*. *ce* is used in each clock cycle to select if a specific message m_i should be encrypted/decrypted or just

be sent out from the design. If the value is high y_{out} is XORed with m_i producing a new message t_i, which is the encrypted/decrypted message. cm, on the other hand, is used to select between decryption and encryption. For decryption, the plaintext will be t_i and m_i will be the ciphertext, while for encryption the opposite is the case. What that means is that there needs to be a mux controlled by cm that select t_i when doing decryption and select m_i when doing encryption, as the input message for the accumulator logic. In the design, a high value on cm is used for decryption and a low value for encryption. It can be noted for the non-parallelized version that an extra register is required in order to delay the t_i signal, during decryption, in order to ensure it is sent in when a y_{accum}/MAC bit is available. For the parallelized versions the register is not required since the keystream bits and MAC bits are generated in the same clock cycle.

3.1.6 Tag Generation

At the end of authentication when the message has been sent in, the controller opens up the mux for the tag so that the content of the accumulator is sent out as the tag. Otherwise, the controller keeps the mux closed by only sending zeros, which prevent the cipher from leaking the content of the accumulator at every state of the design.

3.2 Critical Path Analysis

When it comes to finding paths [25] for optimization there are a few that can be modified without changing the function or security of the cipher. Those paths are:

- Dn: the maximal delay from any NFSR or LFSR flip-flop to any other NFSR or LFSR flip-flop.
- Dy: the maximal delay from any NFSR or LFSR flip-flop through the y function to the output of the cipher.
- Dya: the maximal delay from any NFSR or LFSR flip-flop through the y function to any accumulator flip-flop.
- Da: the maximal delay from any flip-flop in the authentication section of the cipher to any accumulator flip-flop or output.
- Dyn: the maximal delay from a flip-flop of the NFSR or LFSR through the y function to the first flip-flop of the NFSR. Only active during the `initialization` state.

In Figure 3.3 a representation of the paths is given.

3.2.1 Galois Transformation

Improving the delay Dn, might seem like a difficult task since it cannot be pipelined without stopping the cipher completely. However, according to the theory in Section 2.6.2 it is possible to transform the g function in its original Fibonacci form into its Galois equivalent without affecting the security. The transformation

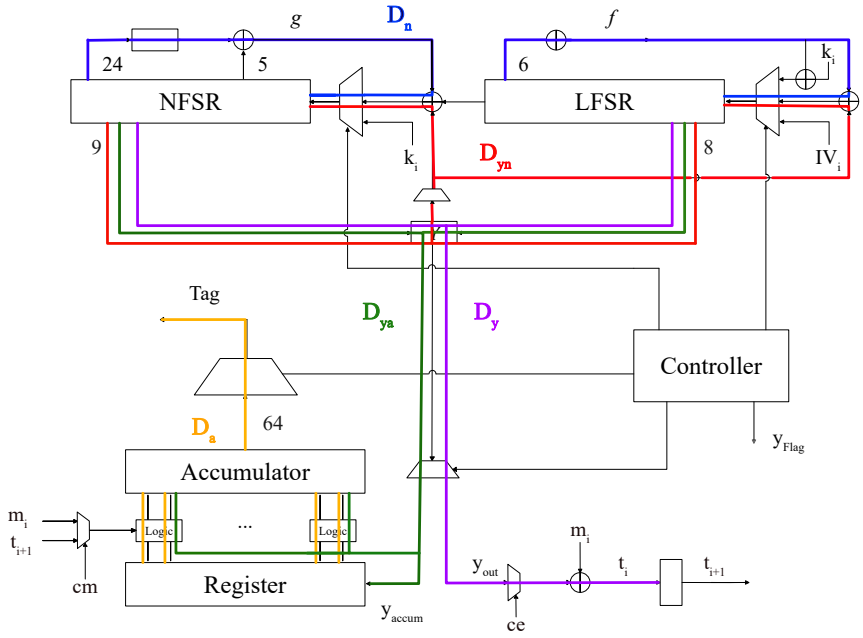


Figure 3.3: A figure showing the possible paths for optimization of the Grain-128AEAD implementation.

will as previously stated decrease the path. Apart from that, there is not much that can be done to improve the path delay, Dn. As previously stated in Section 2.2.3, the 32 parallelized version cannot be transformed into its Galois configuration, which means that the path cannot be optimized.

Using the algorithm found in [21] and [22] the Galois transforms can be generated. The Galois transformation of the NFSR for the non-parallelized version is given as:

$$\begin{aligned}
g_{127} &= s_0 \oplus b_0, & g_{102} &= b_{103} \oplus b_{71},\\
g_{126} &= b_{127} \oplus b_{39}b_{47}, & g_{101} &= b_{102} \oplus b_0,\\
g_{125} &= b_{126} \oplus b_{59}b_{63}, & g_{100} &= b_0 \oplus b_0 b_{32},\\
g_{124} &= b_{125} \oplus b_0 b_{64}, & g_{99} &= b_{100} \oplus b_{63},\\
g_{123} &= b_{124} \oplus b_{52}, & g_{98} &= b_{99} \oplus b_{59}b_{63}b_{64}b_{66},\\
g_{116} &= b_{117} \oplus b_0 b_2, & g_{97} &= b_{98} \oplus b_{38}b_{54},\\
g_{110} &= b_{111} \oplus b_0 b_1, & g_{96} &= b_{97} \oplus b_{39}b_{47}b_{51},\\
g_{105} &= b_{106} \oplus b_0 b_2 b_3, &&
\end{aligned} \tag{3.1}$$

where g_k is the next bit for the NFSR flip-flop at index k. The 2 times

parallelized NFSR is given as:

$$\begin{aligned}
g_{127} &= s_0 \oplus b_0, & g_{111} &= b_{112} \oplus b_6 b_8 b_9 \\
g_{125} &= b_{126} \oplus b_1 b_{65}, & g_{107} &= b_{108} \oplus b_{58} b_{62} b_{50}, \\
g_{123} &= b_{124} \oplus b_{57} b_{61}, & g_{105} &= b_{106} \oplus b_{18} b_{26}, \\
g_{121} &= b_{122} \oplus b_5 b_7, & g_{103} &= b_{104} \oplus b_{72}, \\
g_{119} &= b_{120} \oplus b_9 b_{10}, & g_{101} &= b_{102} \oplus b_{30}, \\
g_{115} &= b_{116} \oplus b_{15} b_{47}, & g_{99} &= b_{100} \oplus b_{40} b_{56}, \\
g_{113} &= b_{114} \oplus b_{12}, & g_{97} &= b_{98} \oplus b_{58} b_{62} b_{63} b_{65}.
\end{aligned} \tag{3.2}$$

The 4 times parallelized NFSR is given as:

$$\begin{aligned}
g_{127} &= s_0 \oplus b_0 \oplus b_3 b_{67}, \\
g_{123} &= b_{124} \oplus b_{22} \oplus b_{52} \oplus b_{23} b_{55}, \\
g_{119} &= b_{120} \oplus b_9 b_{10} \oplus b_3 b_5, \\
g_{115} &= b_{116} \oplus b_{70} b_{66} b_{58}, \\
g_{111} &= b_{112} \oplus b_6 b_8 b_9, \\
g_{107} &= b_{108} \oplus b_{68} b_{72} b_{73} b_{75}, \\
g_{103} &= b_{104} \oplus b_{72} \oplus b_{37} b_{41}, \\
g_{99} &= b_{100} \oplus b_{40} b_{56} \oplus b_{63} \oplus b_{12} b_{20}.
\end{aligned} \tag{3.3}$$

The 8 times parallelized NFSR is given as:

$$\begin{aligned}
g_{127} &= s_0 \oplus b_0 \oplus b_3 b_{67} \oplus b_{88} b_{92} b_{93} b_{95}, \\
g_{119} &= b_{120} \oplus b_9 b_{10} \oplus b_3 b_5 \oplus b_{32} b_{40} \oplus b_{60} b_{76}, \\
g_{111} &= b_{112} \oplus b_{10} \oplus b_{40} \oplus b_{11} b_{43} \oplus b_{75} \oplus b_6 b_8 b_9, \\
g_{103} &= b_{104} \oplus b_{72} \oplus b_{37} b_{41} \oplus b_{46} b_{54} b_{58}.
\end{aligned} \tag{3.4}$$

The 16 times parallelized NFSR is given as:

$$\begin{aligned}
g_{127} &= s_0 \oplus b_0 \oplus b_{56} \oplus b_3 b_{67} \oplus b_{11} b_{13} \oplus b_{40} b_{48} \oplus b_{70} b_{78} b_{82}, \\
g_{111} &= b_{112} \oplus b_{10} \oplus b_{75} \oplus b_{80} \oplus b_1 b_2 \oplus b_{11} b_{43} \oplus b_{45} b_{49} \oplus b_{72} b_{76} b_{77} b_{79} \oplus b_{68} b_{52}.
\end{aligned} \tag{3.5}$$

The Galois transform for the 1, 2 and 4 times parallelized LFSR is given as:

$$\begin{aligned}
f_{127} &= s_0 \oplus s_7, \\
f_{123} &= s_{124} \oplus s_{34}, \\
f_{119} &= s_{120} \oplus s_{62}, \\
f_{115} &= s_{116} \oplus s_{69}, \\
f_{111} &= s_{112} \oplus s_{80},
\end{aligned} \tag{3.6}$$

where f_k is the next bit for the LFSR flip-flop at index k. The 8 times parallelized LFSR is given as:

$$\begin{aligned}
f_{127} &= s_0 \oplus s_7 \oplus s_{38}, \\
f_{119} &= s_{120} \oplus s_{62}, \\
f_{111} &= s_{112} \oplus s_{65}, \\
f_{103} &= s_{104} \oplus s_{72}.
\end{aligned} \tag{3.7}$$

Finally, the 16 times parallelized LFSR is given as:

$$\begin{aligned} f_{127} &= s_0 \oplus s_7 \oplus s_{38} \oplus s_{70}, \\ f_{111} &= s_{112} \oplus s_{65} \oplus s_{80}. \end{aligned} \tag{3.8}$$

3.2.2 Isolation of Authentication Section

In the parallelized versions of Grain-128a the expression for updating the accumulator, Eq. 2.14, requires future values of r_i that are not yet shifted into the shift register in order to update the first few accumulator flip-flops. What that means is that the path Dya for the parallelized versions, $n > 2$, become longer than that of the non-parallelized version. The problem can be resolved by isolating the authentication section from the y function with flip-flops at the cost of an extra clock cycle of latency for the tag.

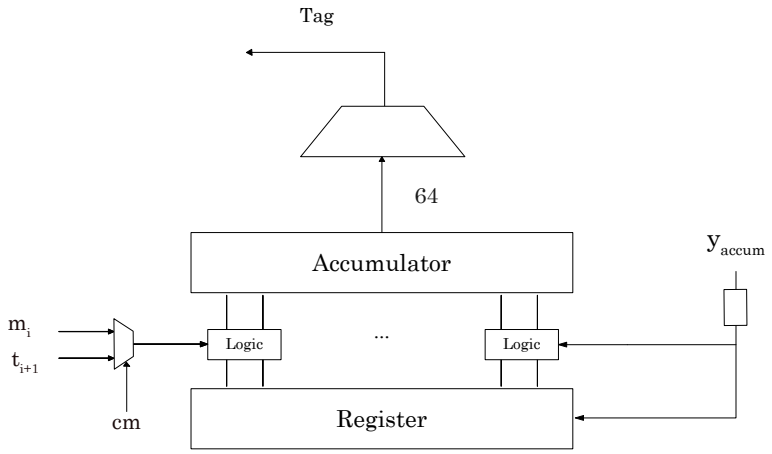


Figure 3.4: An overview of the isolation of the authentication state for the $n > 2$ parallelized versions.

3.2.3 Transforming and Pipelining Y

In order to reduce Dy and Dyn, two methods can be applied: pipelining y and transforming y. During the `initialization` state, pipelining is not possible since y needs to be feedbacked to the shift registers in that state. It is however possible to transform the y function under the same conditions as the Galois transform, which will reduce the paths. For the other states where y is not feedbacked to the shift registers the y transform is not possible, but pipelining is. By combining and selecting between both methods it is possible to realize an implementation that reduces the path Dy and Dyn. A figure of the hardware can be seen in Figure 3.5.

As an example, the transformation of the y function for the non-parallelized

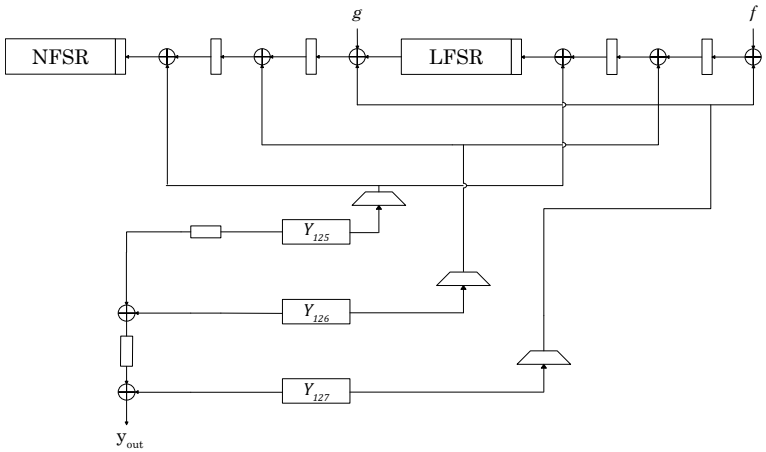


Figure 3.5: An overview of y transformation and pipelining [1].

version from the figure is given as:

$$\begin{aligned} y_{127} &= b_{12}s_8 \oplus s_{13}s_{20} \oplus b_{95}s_{42}, \\ y_{126} &= b_{11}b_{94}s_{93} \oplus b_{72} \oplus b_1 \oplus s_{59}s_{78}, \\ y_{125} &= s_{91} \oplus b_{87} \oplus b_{13} \oplus b_{34} \oplus b_{43} \oplus b_{62}, \end{aligned} \tag{3.9}$$

where y_{125+i}, $0 \leq i \leq 2$, indicates the value that is fed back to s_{125+i} and b_{125+i}. It is possible to design the y transform for the 2-16 parallelized version. Because of similar restrictions as for the Galois transformation, it is not possible to transform y for higher parallelized versions than 16. Note that apart from adding both the transformation and pipelining hardware, muxes are also required in order to switch between both methods. The switching of the methods is regulated by the controller.

3.3 Modifying the Controller

After improving most of the path in Section 3.2, the only other improvements to be made is to make sure as little logic as possible is used to control the design in order to further increase the performance. In this section, an alternative controller is represented using a clock divider and shift register in order to improve area and speed. The controller functions by shifting in a high value into a shift register every time its clock is high and after a start signal has been received from outside the design. A clock divider is used to slow down the shift register clock signal by 2^k, where k is a positive number, making it so the register shifts at a lower speed. This design allows for a bit in the shift register to represent how many clock cycles have passed in total and thus the state of the design, by simply checking if a flip-flop value is high at a certain index. What that means is that only a clock divider circuit is required for updating the state and a maximum of 1-3 bits are required in order to generate the control signals for the multiplexers in the design. The number

of flip-flops required for the shift register depends on both the parallelization, n, and the value of k as $512/(2^k n)$ where 512 is the number of clock cycles required for the non-parallelized version to get to the `normal` state without a clock divider. Adding that together with the k flip-flops from the counter makes the total amount of flip-flops required as $512/(2^k n) + k$. Note that it is not possible to select a $2^k n$ larger than 128 since that would mean that each shift takes longer than 128 clock cycles making it impossible to determine when to enter and leave the `loading` state. A figure of the controller can be seen in Figure 3.6 where Mux Ctrl refers to the signal used to control the muxes. The input to the control logic are the bits of the shift register that become one during the start of `initialization`, `accumloading`, and the `normal` state respectively. Their indexes are calculated as

$$\begin{aligned} i_{\text{init}} &= \frac{128}{nk}, \\ i_{\text{accum}} &= \frac{128 + 256}{nk}, \\ i_{\text{norm}} &= \frac{512}{nk}. \end{aligned} \tag{3.10}$$

The control logic is different from muxes to muxes. For muxes that remain open after they are first turned on, the signals can be directly ported to the multiplexers. However, for other signals that are only activated in one state an inverter together with an AND-gate is required to generate the mux control signal.

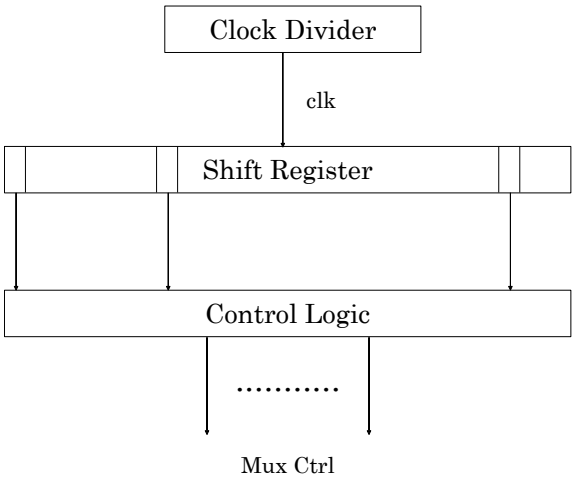


Figure 3.6: An overview of the modified controller.

3.4 Unrolling

Unrolling [26] is a general technique that has been discussed already in Section 2.2.3 to increase the throughput of a design. However, in the section, it is referred to as parallelization and it is stated that a parallelization beyond 32 is not possible using the method described in the section. Fortunately, it is possible to realize a parallelization or unrolling larger than 32 by simply allowing connections between the copies of the feedback and output functions. This can be confirmed by looking at Eq. 2.8 with a k higher than 31. If k is equal to 32 the value b_{i+96+k} in the 33rd g function will be equal to b_{i+128}, which is the bit from the first g function. The downside of allowing these types of connections is that the critical path will rapidly increase with the number of unrolls after 32. This does however not necessarily mean that the maximum throughput of the design will decrease with unrolling.

Apart from speed, unrolling is also an effective method for saving energy. A n times unrolled cipher compared to the non-parallelized version saves energy by allowing the stream cipher to encrypt a larger amount of data in each clock cycle so that fewer glitches and transitions occur.

A figure of the implementation of unrolling for the NFSR at $k = 32$ can be seen in Figure 3.7, where g_0 refers to the first feedback function and g_{32} the last feedback function. For $k > 32$ the concept is very similar, with there only being more connection to the previous function for the y, g and f functions respectively. In the thesis, only the 64 unrolled version is implemented. It is possible to implement other unrolled versions, but that would require a controller capable of being in two states at once.

It is also possible to unroll the accumulator with a higher number than 16 using Eq. 2.14 assuming that the Grain-128AEAD parallelization is twice as high.

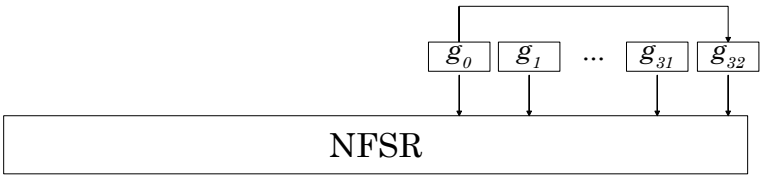


Figure 3.7: A figure showing the concept of unrolling for the g function at $n = 33$.

Chapter 4
Synthesis

In the previous chapter, we presented different implementations and techniques for the optimization of Grain-128AEAD at RTL level, which all improve the cipher functionality at hardware level. The following chapter will cover the different possibilities for synthesizing the RTL code in the Synopsys Design Compiler 2013.12 with an ST65nm ASIC technology.

4.1 Cipher Synthesis

Synthesis [27] is a process that uses hardware description language to generate a gate-level net-list for the circuit designer. It consists of three important steps: Translation, Optimization and Technology Mapping.

- Translation: The RTL code is converted to technology-independent representation, in the form of a Boolean expression.
- Minimization: The Boolean expressions are minimized using the SoP (Sum of Product) or PoS (Product of Sum) method.
- Technology mapping: The optimized Boolean expressions are mapped to the technology-dependent library file to produce a gate-level netlist.

4.1.1 Conditions and Restrictions

Since we are going to compile our RTL code in Design Vision some operating conditions needs to be set in the tool.

The first phase of synthesis is to analyze and elaborate [28] the RTL files which will covert the HDL to a readable format for the synthesis tool.

After this step, the design environment is set for the compiler by specifying the search path, link library, target library, symbol library and synthetic library for the specific technology, pads, etc. in the Synopsys tool.

The search path is the path to the technology-specific library. Whereas, the link library and target library are the files that contain the cells as well as some additional information related to the cells, such as its operating conditions, pin names, and delays, etc. In addition to that, the link library also helps in connecting all the library components and designs, while the target library helps in the mapping of the standard cells.

Moreover, the symbol library is the graphic symbols used by the library to represent its cell in a graphical interface, which allows for the schematic view of a synthesized circuit.

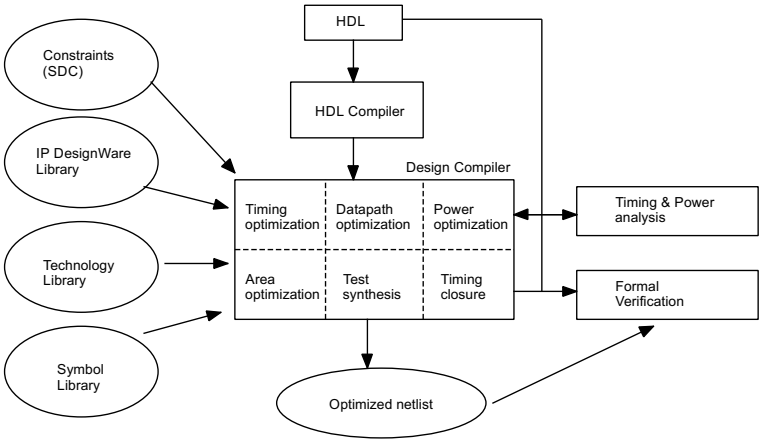


Figure 4.1: Synopsis DC flow diagram.

4.1.2 Compiling or Mapping the Design

The next phase of synthesis is compiling and mapping the design. Design Vision [29] needs three different inputs to compile the cipher and generate the result. These inputs are the RTL code, design constraints, and standard library cell. There are two commands which can be used for compiling the RTL files in Design Vision `compile` and `compile_ultra` [30]. Both commands have lots of flags that can be set to optimize for power, speed or area. Some of the flags which were used in writing the synthesis script include clock gating, which enables the clock gating technique to reduce power in the circuit; grouping that helps in creating the hierarchy level of cell and optimize the design for speed; Ungrouping which removes the level of hierarchy and merge with the surrounding logic to reduce the area.

4.1.3 Reporting the Results

This is the last step in the process of synthesis which helps in generating the reports for the optimized results. Design Vision can generate different reports through command line [31] or by click on the toolbar options. There are a lot of commands to generate reports [32] but we were interested in some specific information for the design of the cipher at different frequency range like timing (it helps in finding the worst critical paths and slack), power consumption, clock, constraint (to determine whether DC encountered any violations), area and gate-count.

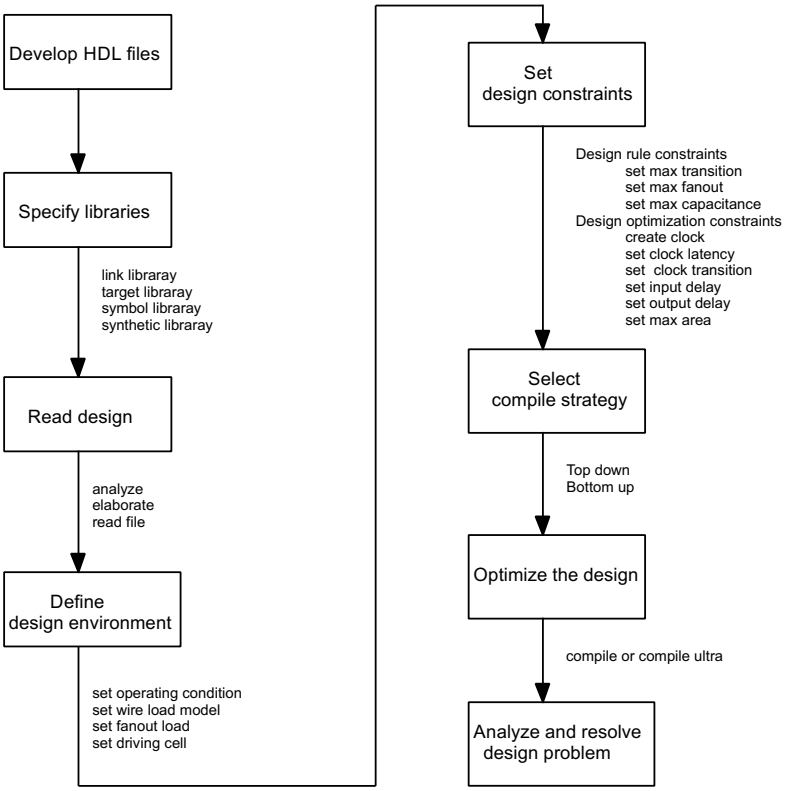


Figure 4.2: Flow diagram for optimizing Grain128-AEAD.

4.2 Synthesis Scripts

Since the cipher is to be implemented for different frequency ranges and restrictions, two different scripts Appendix A are therefore written for optimizing at low power/area and high speed. The first script is written for low power/area by using HVT cells and enabling the clock gating feature in `compile_ultra` command in the script to reduce the power consumption of the circuit. The second script is written to run the cipher for maximum speed without caring about power/area. In the latter script, the `compile_ultra` command is used with and without the autoungroup feature with the combination of LVT cells for optimizing the cipher to run as fast as possible.

Chapter 5

Transistor Optimizations

Today, digital circuits are mainly driven by demand for high processing speeds and low power applications. Metal Oxide Semiconductor (MOS) transistors are commonly used as switches with high or low-speed activities. Since the circuit topologies that are presented in this thesis are based on threshold voltage of MOS devices, a brief overview of the MOSFET and optimization technique to reduce leakage in the CMOS device is provided in this chapter. Later on, there is a discussion about the Boolean expression optimization for reducing the area, power and improving the speed of the ciphers.

5.1 MOSFET

Moore's law [33] explains that the transistors count in an electronics circuit will double about every two years. So moving a design from a longer length of MOSFET to a shorter length, will always be an interesting way to mitigate the power consumption in the circuits.

5.1.1 Region of Operation of MOS Transistor

Transistors can operate in different regions of operation [34] which are saturation and linear regions refer in Figure 5.1, depending on the voltage applied to the gate(G), drain(D) and source(S). There are two type of transistors NMOS and PMOS. In NMOS, the majority carriers are electrons, and they can flow faster than holes. As a result, NMOS transistors are smaller than PMOS devices. The majority carriers in PMOS are holes. Holes flow more slowly compared to electrons, therefore it is easier to control the current. The combination of both transistors is called complementary-symmetry metal–oxide–semiconductor (or CMOS), which has high noise immunity and low static power consumption which will be discussed next section.

5.1.2 Techniques to Reduce Leakage Current

Circuits are designed for performance and they are designed using large gates and parallel architecture with the same logic. Actually, to improve the performance

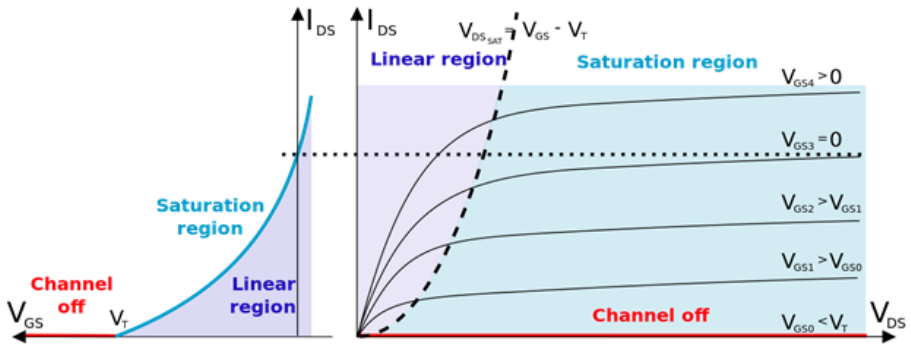


Figure 5.1: Transistor region of operation.

of the digital circuit the overall parasitic capacitance (i.e., gates and interconnects) and the supply voltage should be decreased. Moving to a small MOSFET size reduces the supply voltage (V_{dd}), the threshold voltage (V_{th}), and the gate oxide thickness (T_{ox}). There are three types of the transistors depending upon threshold voltage: Low Threshold Voltage (LVT) transistor that are used for high performance, Standard Threshold Voltage (SVT) transistors and High Threshold Voltage (HVT) transistor use for low leakage.

There are different techniques that have been proposed to reduce the leakage energy without affecting the performance. These techniques can be divided based on the available time slack. They are called design time techniques and run time techniques. Design time technique is used for non-critical paths which help in reducing leakage with Multi-Supply Voltage [35] and Dual V_{th} [36] technique. They are static and cannot be changed while the circuit is operating. Run time techniques can be sub-divided into two groups depending on the reducing standby leakage or active leakage. Standby leakage puts the system in low leakage mode when the operation is not required with support of sleep transistors [37], stacking [38] and VTCMOS [39]. Active leakage makes the system slower by changing the V_{th} or V_{dd} when performance is not required, while reducing the leakage with the DVS [40] and DVTS [41] technique.

Table 5.1: Optimization technique for leakage current.

Design time techniques	Run time techniques	
	Standby leakage reduction	Active leakage reduction
Dual V_{th}	Sleep transistor	DVS (Dynamic V_{dd} Scaling)
Multi-Supply Voltage	Stacking and VTCMOS	DVTS (Dynamic V_{th} Scaling)

5.1.3 Techniques for Optimization Cipher Expressions

Circuit optimization is a crucial part of designing high-performance circuit. Optimization is an automated process for achieving the performance of the circuit which is already done in the previous chapter. In this section the main focus is on Boolean expression optimization at transistor level which will later be designed in cadence to get proper functionality of the circuit.

Boolean Algebra

Boolean algebra [42] is a set of rules which are developed to solve or reduce the number of logic gates required to perform some particular digital logical functions. It is mathematics which help in the analysis of digital gates and circuits. There are lots of different laws of Boolean algebra like Annulment Law, Identity Law, Idempotent Law, Double Negation Law, De Morgan's theorem (Figure 5.2), etc. In Grain-128AEAD [1] there are complex Boolean expressions which are designed using XOR and XNOR gates, which cannot be optimized using Boolean algebra at a transistor level, therefore, there are different design techniques for complex Boolean expressions [43]. In our thesis, we will analyze pass-transistor logic and transmission gate logic for transforming the complex expressions to other Boolean expressions that use fewer transistors. They are explained in the following sections.

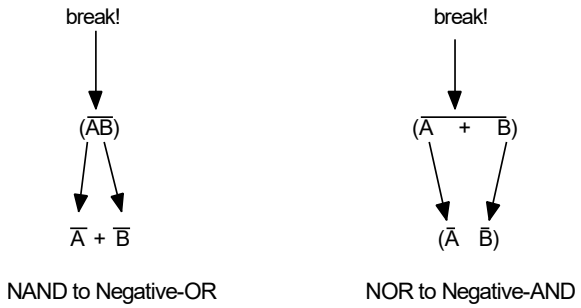


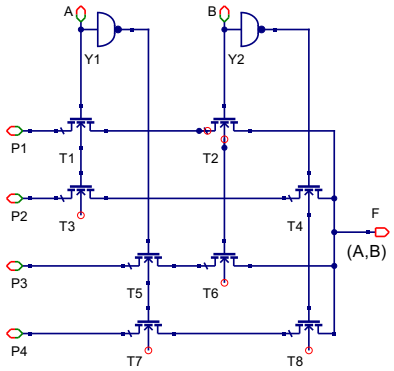
Figure 5.2: De Morgan's theorem.

MOSFET as a Pass Gate logic

The pass transistor [44] design has the advantage of being simple and fast. Complex combinational logic can be implemented with a minimal number of transistors, which will reduce parasitic capacitance. Hence, the circuit will work faster. As a pass transistor design example, Figure 5.3 shows a Boolean function realized using pass transistors. In this circuit the F output is either the NOR ($\overline{A+B}$), XOR ($A \oplus B$), NAND ($\overline{AB}$), AND (AB) or OR (A+B) depending on the values of P1, P2, P3 and P4.

One of the main drawbacks of the pass transistor implementation is the voltage drop when signals pass through them. Another is the high internal capacitance because the junction capacitors are open to the signals passing through. Therefore to overcome this problem each pass gate based circuit should follow with an active

logic block(level restorer), such as a CMOS inverter support with a full swing PMOS (as shown in below Figure 5.4).



P1	P2	P3	P4	F(A,B)
0	0	0	1	NOR
0	1	1	0	XOR
0	1	1	1	NAND
1	0	0	0	AND
1	1	1	0	OR

Figure 5.3: Standard gate design using pass transistor.

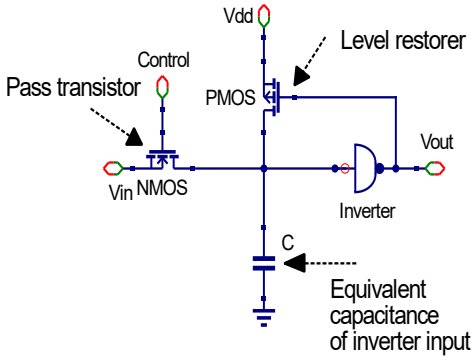
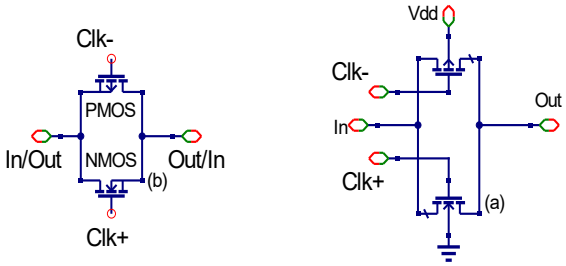


Figure 5.4: Level restorer.

MOSFET as a Transmission Gate Logic

The transmission gate [45] operates as a bi-directional switch. It requires two control signals to control NMOS and PMOS. Through transmission gates, we can design XOR/XNOR, latches, multiplexers and flip-flops. The schematic and truth table is shown in Figure 5.5. The control signal A is applied to the NMOS and the complement of the control signal $\overline{S}$ to the PMOS. If the control signal A is high, both transistors are turned on providing a low resistance path between IN and OUT. If the control signal A is low, both transistors will be off and the

path between the nodes IN and OUT will be an open circuit. There are lots of advantages of a transmission gate. A complex gate can be implemented using a minimum number of transistors using a transmission gate. The combination of both NMOS and PMOS in the transmission gate avoids the problem of static power dissipation, noise margin and switching resistance.



(C)

Clk	nMOS	pMOS	Output
0	OFF	OFF	Z(impedance)
1	ON	ON	INPUT

Figure 5.5: Transmission Gate circuit diagram and symbol.

5.2 Boolean Expression Optimization for Grain-128AEAD

All the optimization of the cipher [1] in the Synopsys tool is completed using a standard cell library. Later on, while analyzing the synthesis output in Design Vision different circuit were there for different functions of the cipher.

The general criteria to convert any Boolean expression to transistor level is to design the truth table and analyze the expression using the K-map method. The second step is to design a Pull Up Network (PUN) and a Pull Down Network (PDN) circuit for the Boolean expression like shown in Figure 5.6. For "A AND B" the PDN network will have two transistors in series and PUN will have two transistors in parallel. Similarly, for "A OR B" the PUN network will be in series and a PDN network will be in parallel.

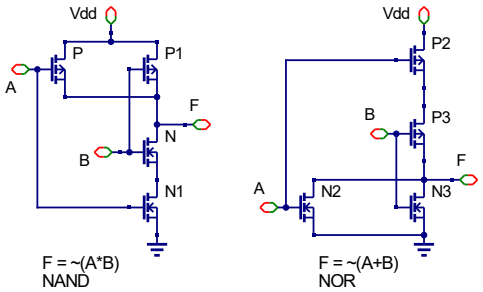


Figure 5.6: Boolean expression to transistor level.

The cipher expressions that are going to be optimized are complex Boolean expressions, therefore, the general Boolean algebra technique is not applicable when it is optimizing for low power, less transistor count or high speed. We need to refer to complex circuit optimization techniques which are mentioned in the previous sections. During the analysis of different functions of Grain-128AEAD, the first optimization is applied to y function, which has a duplicate variable in its expression and it looks similar to "AB $\oplus$ ACD". Therefore, we tried to optimize the y using a transmission gate technique for reducing the transistor count and improving the functionality of the expression as seen in Figure 5.7.

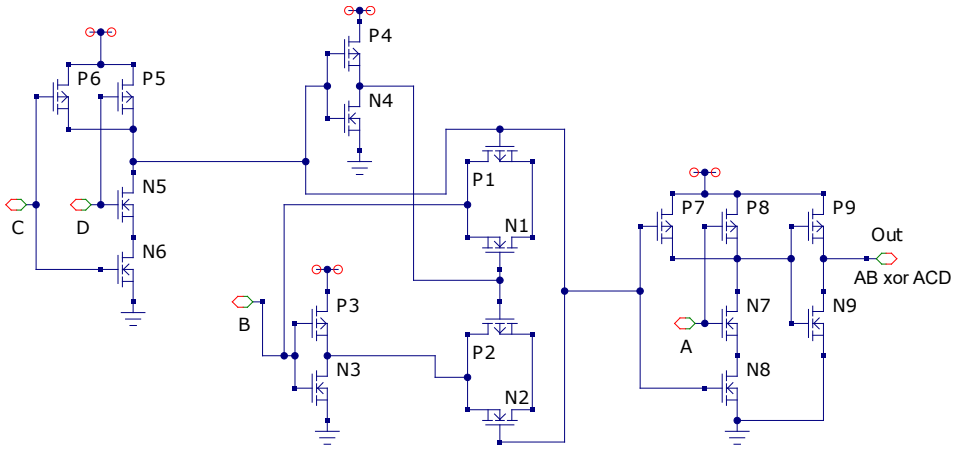


Figure 5.7: A figure of a boolean expression "AB xor ACD" optimized at transistor level.

The second optimization is attempted for the nonlinear feedback polynomial function which uses a polynomial of two, three and four variable. Hence, all the Boolean expressions like "AB $\oplus$ CD $\oplus$ EF $\oplus$ GH" (Figure 5.8), "AB $\oplus$ CDE", "CDE $\oplus$ FGH" etc. are designed using transmission gate technique to improve the circuit performance for the battery operated embedded security devices. In Appendix B additional transistor optimizations can be seen for the above mention Boolean expression.

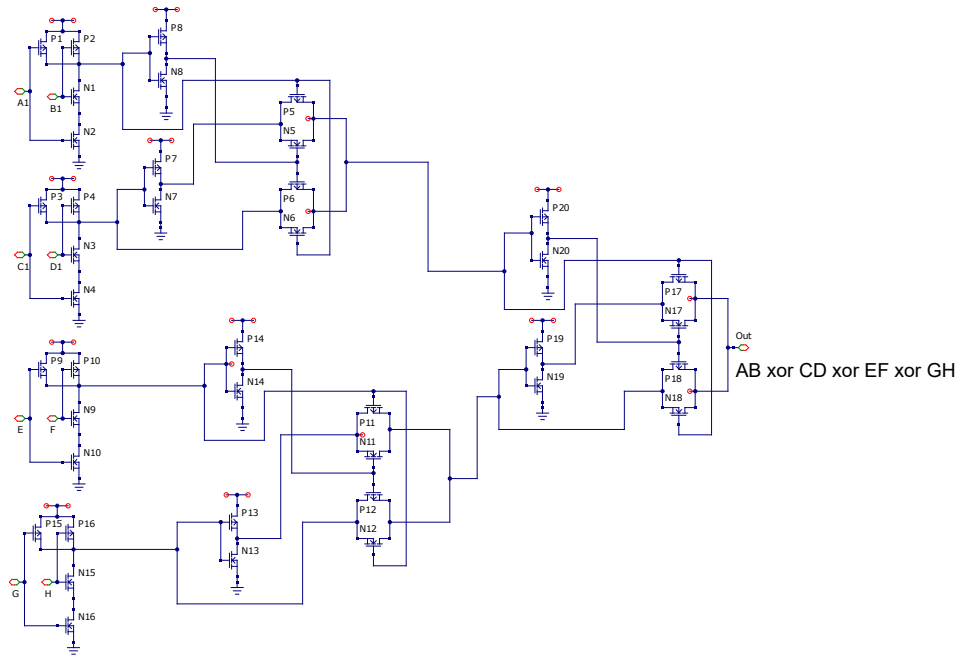


Figure 5.8: A figure of a boolean expression "AB xor CD xor EF xor GH" optimized at transistor level.

Chapter 6

Result & Conclusion

The result is divided into two groups: straightforward implementations and optimized implementations. In each group we use the scripts presented in Chapter 4 to obtain one circuit optimized for maximum speed and another circuit optimized for low area and power. The high-speed circuit uses two different scripts that we refer to as synthesis script 2 and 3, where the script that gives the best result for each parallelization is listed in the result. Synthesis script 2 is `compile_ultra` with grouping and synthesis script 3 is without grouping as is described in Section 4.1.2. The low area/power circuit is synthesized using the best high-speed script and the low power/area script for comparison between the scripts. The low power/area script, which we refer to as synthesis script 1 includes `compile_ultra` with clock gating. The high speed script uses high-speed transistors (LVT) and the low power/area script uses low-power transistors (HVT). Moreover, the result for the low power/area script is given in two frequencies: 100 kHz and 10 MHz.

6.1 Straightforward Implementation

In the straightforward implementation, no RTL optimizations are used, which makes it a good candidate for low area and power applications compared to the optimized versions that all use extra hardware to increase throughput. The maximum speed is however still investigated in order to get a comparison with the optimized implementation.

6.1.1 High Speed

The result for the high-speed circuit of the straightforward Grain-128AEAD implementation for all parallelizations are presented in Table 6.1 using the standard state machine and counter controller.

To see how the modified controller performs in comparison to the normal controller for the straightforward implementation, the same result is presented using the modified controller in Table 6.2

It can be noted from the result that the new controller seems to perform significantly better in comparison to the standard controller. The area is smaller and the power consumption is even decreased for the 32-parallelized version even though the cipher is running at a higher maximum frequency.

Table 6.1: High speed result for the straightforward implementation.

n	Max Per. (ns)	Max Freq. (GHz)	Throughput (Gb/s)	Area (μm^2)	Power (mW)	Script
1	0.49	2.04	1.02	5594	0.17	3
2	0.61	1.64	1.64	5774	0.14	2
4	0.64	1.56	3.12	6932	0.21	2
8	0.69	1.44	5.76	8994	0.42	2
16	0.77	1.29	10.32	13032	0.92	2
32	0.84	1.19	19.04	21271	2.54	2

Table 6.2: High speed result for the straightforward implementation using the modified controller.

n	Max Per. (ns)	Max Freq. (GHz)	Throughput (Gb/s)	Area (μm^2)	Power (mW)	Script
1	0.48	2.08	1.04	5219	0.18	3
2	0.49	2.04	2.04	5498	0.19	2
4	0.51	1.96	3.92	6385	0.28	2
8	0.56	1.78	7.12	8056	0.57	2
16	0.63	1.59	12.72	11966	0.91	2
32	0.73	1.37	21.92	19047	1.62	2

6.1.2 Low Power and Area

The result of running the low area/power script (synthesis script 1) compared to the best high-speed script at 10 MHz and 100 kHz is given in Table 6.3 and Table 6.4. In the table, the area/power script is denoted PS and the speed script is denoted SS. The SS script used for each parallelization is the one that gave the best result and is listed in the table under script. Moreover, a result when using the modified controller with the area/power script is also included in the table, denoted as PS2, for comparison.

It is also worth mentioning that the energy per bit in the table is estimated from the power result as TP/n where T is the clock period and P is the power result from the table.

Overall the area and power are decreased as expected for lower frequencies. By comparing the power/area script and high-speed script, it is clear that the low power/area script is superior when it comes to power. However, the area did not improve very drastically between the two, but still improved slightly. It is safe to assume that the reason for the slight improvement in the area is because there is not much hardware that can be changed or removed without altering the function of the design at such low frequency.

From the power result, it is clear that the non-parallelized version consumes

Table 6.3: Result for the straightforward implementation running at 100 kHz with the speed script compared to the power/area script using the standard and modified controller.

n	Area (μm^2)			Power (μW)			Energy/bit (pJ)			Script
	SS	PS	PS2	SS	PS	PS2	SS	PS	PS2	SS
1	5218	4939	4861	2.29	0.23	0.26	22.9	2.3	2.6	2
%	-	-5	-7	-	-89	-88	-	-89	-88	-
2	5391	5383	5222	2.33	0.28	0.30	11.6	1.4	1.5	2
%	-	0	-3	-	-87	-86	-	-88	-87	-
4	6141	6137	5952	2.33	0.29	0.32	5.8	0.72	0.81	2
%	-	0	-3	-	-87	-86	-	-87	-86	-
8	7686	7679	7475	2.76	0.31	0.35	3.4	0.38	0.44	2
%	-	0	-2	-	-88	-87	-	-88	-87	-
16	10749	10729	10511	3.77	0.42	0.39	2.3	0.26	0.24	2
%	-	0	-2	-	-89	-90	-	-88	-89	-
32	16990	16903	16537	5.93	0.62	0.46	1.8	0.19	0.14	2
%	-	0	-3	-	-90	-92	-	-89	-92	-

Table 6.4: Result for the straightforward implementation running at 10 MHz with the speed script compared to the power/area script using the standard and modified controller.

n	Area (μm^2)			Power (μW)			Energy/bit (pJ)			Script
	SS	PS	PS2	SS	PS	PS2	SS	PS	PS2	SS
1	5219	4939	4861	33.66	22.07	25.21	3.3	2.2	2.5	2
%	-	-5	-6	-	-34	-25	-	-33	-24	-
2	5391	5384	5222	33.96	26.93	29.13	1.6	1.3	1.4	2
%	-	0	-3	-	-21	-14	-	-18	-12	-
4	6141	6139	5958	34.43	27.38	31.05	0.86	0.68	0.77	2
%	-	0	-3	-	-20	-10	-	-20	-10	-
8	7686	7682	7477	36.83	29.38	33.59	0.46	0.36	0.4	2
%	-	0	-3	-	-20	-9	-	-21	-13	-
16	10749	10737	10518	44.02	39.49	36.93	0.27	0.24	0.23	2
%	-	0	-2	-	-10	-16	-	-11	-14	-
32	16897	16907	16535	66.02	57.08	41.66	0.20	0.17	0.13	2
%	-	0	-2	-	-13	-37	-	-15	-35	-

less power, as expected. However, since higher parallelized versions compared to lower parallelized versions allow for more bits to be generated in each clock cycle, the consumed energy per generated output bit is actually decreased. What that means is that in situations where the cipher is not running all the time, it can be more energy efficient to use the higher parallelized versions. The biggest save in terms of energy comes at 86-92% running at 100 kHz and 10-35% at 10 MHz.

It can also be noted that the modified controller outperforms the standard version in terms of area. For power, however, the standard controller beats the

modified controller for the 1-8 parallelized versions, while the opposite is true for 16-32. The version that benefits the most is the 32 version with a 92% and 37% power improvement on 100 kHz and 10 MHz respectively. The versions that benefit the least but still plenty is the 2-parallelized and 4-parallelized versions with an 86% power improvement at 100 kHz and the 8-parallelized version with an improvement of 9% on power at 10 MHz.

6.2 RTL Optimizations

In an attempt to improve the result in the previous section the optimizations presented in Section 3.2 are applied to the straightforward implementation. For the 1, 2, 4, 8 and 16 parallelization version the Galois transformation, y transformation and the isolation of the authentication state is performed. However, for the 32 and 64 versions, only the isolation of the authentication state optimization is possible, thus the two other optimizations are not included in those versions.

6.2.1 High Speed

The result for running the high speed script at maximum frequency can be seen in Table 6.5 and Table 6.6.

Table 6.5: High speed result for the optimized implementation.

n	Max Per. (ns)	Max Freq. (GHz)	Throughput (Gb/s)	Area (μm^2)	Power (mW)	Script
1	0.43	2.3	1.15	5806	0.24	3
2	0.46	2.17	2.17	5824	0.21	2
4	0.47	2.13	4.26	6936	0.29	2
8	0.48	2.08	8.32	9436	0.67	2
16	0.50	2	16	13042	1.44	2
32	0.69	1.45	23.2	19028	2.66	2
64	1.0	1	32	34565	4.76	2

As can be seen from the result, the 64 unrolled version gives the highest throughput, area and power consumption. This result is very interesting since a 64 unrolled versions of Grain-128a and Grain-128AEAD have not up to this point been considered before.

Overall the optimized version with the modified controller gives the best result in terms of throughput with 33.6 Gb/s for the 64-parallelized, being the highest throughput achievable. It is safe to say that the optimized modified controller is the better version when it comes to speed.

Table 6.6: High speed result for the optimized implementation using the modified controller.

n	Max Per. (ns)	Max Freq. (GHz)	Throughput (Gb/s)	Area (μm^2)	Power (mW)	Script
1	0.4	2.5	1.25	5501	0.25	3
2	0.43	2.32	2.32	5605	0.23	2
4	0.48	2.08	4.16	6654	0.29	3
8	0.46	2.17	8.68	9252	0.67	2
16	0.48	2.08	16.64	14805	1.55	3
32	0.64	1.56	24.96	19149	1.78	3
64	0.95	1.05	33.6	35272	2.76	3

6.2.2 Low Power and Area

For comparison to the straight forward implementation, the result from running the low area/power script for the optimized version is presented in Table 6.7 for 100 kHz and in Table 6.8 for 10 MHz. In the table OPS and OPS2 is denoted as the result for running the low area/power script on the standard controller and the modified controller respectively. Moreover, the best result from Table 6.3 and Table 6.4 is included for comparison. The best result from the straight forward implementation is specified in the table under script as either 1 for PS or 2 for PS2.

Table 6.7: Result for the optimized implementation running at 100 kHz with the power/area script for the standard controller and the modified controller compared to the best result from the straightforward implementation.

n	Area (μm^2)			Power (μW)			Energy/bit (pJ)			Script
	PS	OPS	OPS2	PS	OPS	OPS2	PS	OPS	OPS2	PS
1	4939	5173	4968	0.23	0.29	0.29	2.3	2.9	2.9	1
%	-	4	0	-	27	28	-	26	26	-
2	5383	5534	5391	0.28	0.29	0.30	1.4	1.4	1.5	1
%	-	3	0	-	2	5	-	0	7	-
4	6138	6378	6232	0.29	0.30	0.32	0.7	0.7	0.8	1
%	-	4	2	-	4	9	-	0	14	-
8	7679	8153	7988	0.31	0.34	0.35	0.4	0.4	0.4	1
%	-	6	4	-	8	12	-	0	0	-
16	10510	11556	11385	0.39	0.43	0.41	0.2	0.3	0.3	2
%	-	10	8	-	8	4	-	50	50	-
32	16537	17371	17148	0.46	0.64	0.51	0.1	0.2	0.1	2
%	-	5	4	-	40	12	-	100	0	-
64	28705	30247	29938	0.63	1.08	0.74	0.09	0.2	0.1	2
%	-	5	4	-	72	18	-	122	11	-

Table 6.8: Result for the optimized implementation running at 10 MHz with the power/area script for the standard controller and the modified controller compared to the best result from the straightforward implementation.

n	Area (μm^2)			Power (μW)			Energy/bit (pJ)			Script
	PS	OPS	OPS2	PS	OPS	OPS2	PS	OPS	OPS2	PS
1	5209	5173	4968	22.07	29.62	28.57	2.2	2.9	2.8	1
%	-	0	-4	-	34	29	-	31	27	-
2	5383	5538	5394	26.93	27.58	28.43	1.3	1.3	1.4	1
%	-	3	0	-	2	6	-	0	7	-
4	6139	6380	6237	27.38	28.43	30.05	0.6	0.7	0.8	1
%	-	4	2	-	4	10	-	16	33	-
8	7682	8157	7992	29.38	31.70	32.86	0.36	0.39	0.41	1
%	-	6	4	-	8	12	-	8	14	-
16	10516	11565	11388	36.93	39.80	38.18	0.23	0.24	0.23	2
%	-	10	8	-	-0.4	3.4	-	4	0	-
32	16534	17369	17147	41.66	59.93	47.14	0.13	0.18	0.14	2
%	-	5.0	4	-	44	13	-	38	8	-
64	28728	30265	29956	55.93	100.6	67.23	0.08	0.15	0.1	2
%	-	5	4	-	80	20	-	87	25	-

As can been seen from the results of the tables, the power consumption and area does not get better using the optimized RTL version, which makes the straightforward implementation the winner in terms of power and area.

6.3 Transistor Level Optimizations

To get an estimation of how much the transistor level optimizations improve the design, we need to look at the gate and transistor count for the whole design. However, in order to do that we must first look at the transistor count and the gate count of each cell. Table 6.9 contains the transistor count and gate count for a few different gates in ST65nm standard cell library. The gate count of every gate is calculated as its transistor count divided by the transistor count for NAND (4). All transistor counts are retrieved from the ST65nm cell libraries. The result for the original cipher without any transistor optimization is denoted as ORG in the table, while the cipher with the transistor optimizations is denoted as OPT.

The transistor-level optimizations are made only on expressions with AND and XOR gates, which are changed to NAND and XNOR gates by the synthesis tool. So, in order to estimate how many gates can be saved with the transistor level optimizations we consider the feedback and output functions as NAND and XNOR gates. Since the optimizations can only be studied in terms of area due to the difficulty of implementing a custom library used for synthesis, the standard modified controller at 100 kHz, which has the best area, is used for calculating the gate count of all the control/extra logic. In Table 6.10, the gate count for design before and after optimization is displayed. The result before optimization is calculated using Table 6.9. The result after optimization is estimated using the

Table 6.9: Transistor count and gate count for the ST65nm technology.

ST65nm		
Function	Transistor Count	Gate Count
Inverter	2	0.5
AND/OR2	6	1.5
AND/OR3	8	2
AND/OR4	10	2.5
NAND/NOR2	4	1
NAND/NOR3	6	1.5
NAND/NOR4	8	2
XOR/XNOR2	10	2.5
DFF	32	8

same table with the difference being that XNOR is worth 2 instead of 2.5 in the gate count. This change comes from the fact that the transmission gate acting as an XNOR gate saves 2 transistors.

Table 6.10: Gate count for Grain-128AEAD without transistor level optimization compared to with transistor level optimization.

Gate Count	Parallelization													
Building Block	1x		2x		4x		8x		16x		32x		64x	
Version	ORG	OPT	ORG	OPT	ORG	OPT	ORG	OPT	ORG	OPT	ORG	OPT	ORG	OPT
LFSR	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
NFSR	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
f	12.5	10	25	20	50	40	100	80	200	160	400	320	800	640
g	49.5	42	99	84	198	168	396	336	792	672	1584	1344	3168	2688
y	35.5	29.5	71	59	142	118	284	236	568	472	1136	944	2272	1888
Accumulator	512	512	512	512	512	512	512	512	512	512	512	512	512	512
Shift Register	512	512	512	512	512	512	512	512	512	512	512	512	512	512
Accumulator Logic	224	192	224	192	448	384	896	768	1792	1536	3584	3072	7168	6144
Total	3393.5	3347.5	3491	3431	3910	3790	4748	4508	6424	5944	9776	8816	16480	14432
%	-	-1	-	-2	-	-3	-	-5	-	-7	-	-10	-	-12

According to the result 1% to 12% is saved in total. For other ciphers, it might be possible to save more area, but for Grain-128AEAD not much else can be done unless different technologies other than CMOS is considered.

6.4 Analysis

Looking at the design result it can be interesting to analyze how, based on our result, the Grain-128AEAD performs in general. If we take a normal AAA battery with approximately a 1.3 Wh capacity and let it act as a power source for the

Grain-128AEAD. It can be estimated, assuming for simplicity that the capacity is constant, that the most power efficient (standard implementation) can run for 236 to 562 years at 100 kHz and 3 to 7 years at 10 MHz non-stop. Moreover, the fastest implementations (Optimized version with modified controller) can run for 19 to 220 days at full speed non-stop. For more realistic situations where the cipher is not constantly encrypting/decrypting every clock cycle, the result is of course higher.

Looking at techniques, one interesting technique to look at is RFID (Radio Frequency Identification). RFID is a very common technique that plays a large role in the Internet of Things. It uses electromagnetic induction to detect, track, identify and communicate with tags that contain stored data. RFID is used in today's industry for an abundant amount of things like for example assset tracking, ID badging, access control and so on. Unfortunately, most common RFID tags can easily be cloned and read by unauthorized parties [46][47]. For tags that require additional security to prevent these issues it is estimated that for a 5 cent design roughly 500 to 5000 gates can be used as resources for security [48]. Looking at our gate count result from Table 6.10 it seems plausible to fit the lower parallelized versions of Grain-128AEAD onto such a design in order to give the tags authentication and encryption/decryption. There already exist certain designs already that uses Grain-128a, which is very similar to Grain-128AEAD, to add security features. One example being the IT70 Secure Passive RFID Tag [47]. For the lowest cost tags with around 500 gates it is of course not possible to fit Grain-128AEAD since the gate count is too high, but also because the power consumption for such tags are limited to 10 uW at around 10 MHz [49], which is approximately what the cipher itself consumes.

6.5 Conclusion

In conclusion, we have managed to achieve all goals set out for the thesis. We have optimized Grain-128AEAD on an RTL level, transistor level and synthesis level for the area, speed, and power. Giving us the best throughput at 1.25 Gb/s for the non-parallelized version and 33.6 Gb/s for the new 64-parallelized version. The best area improvement of 2-7% for synthesis at 100 kHz and 10 MHz, and 1-12% for the transistor level optimizations. In addition to the best power improvement of 52-94% for the 64-parallelized version and the least improvement of 21% for the 4-parallelized version running on the power script.

When it comes to answering the question of which version is the best in terms of area, speed, and power, the answer is that the 64-parallelized optimized version using the modified controller is the fastest for high speed and the non-parallelized version with the modified controller is the best in terms of area. For power it is differs depending on the parallelization with the 1-8 parallelized versions performing better with the straightforward implementation without the modified controller and the rest performing slightly better with the modified controller when it comes to the 16-64 parallelized versions.

Future work could include making optimizations for other ciphers and see how their result compares to Grain-128AEAD. It could also be to make more unrolled

versions of Grain-128AEAD with a controller that can handle a parallelization which is not in the power of 2.

References

[1] M. Ågren, M. Hell, T. Johansson, and W. Meier, "Grain-128a : a new version of grain-128 with optional authentication," *International Journal of Wireless and Mobile Computing*, pp. 48–59, 2011.

[2] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC press, 2001, ch. 6.

[3] *Linear-feedback-shift-register*, 2019. [Online]. Available: https://patents.google.com/patent/EP0438322A2/en

[4] *NonLinear-feedback-shift-register*, 1990. [Online]. Available: https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_361

[5] A. Chakraborty, B. Mazumdar, and D. Mukhopadhyay, "Fibonacci lfsr vs. galois lfsr: Which is more vulnerable to power attacks?" *International Conference on Security, Privacy, and Applied Cryptography Engineering*, vol. 4, pp. 14–25, 2014.

[6] *Galois configuration*, 2018. [Online]. Available: https://www.gaussianwaves.com/tag/galois-lfsr/

[7] M. Hell, T. Johansson, A. Maximov, and W. Meier, "A stream cipher proposal: Grain-128," *IEEE International Symposium on Information Theory*, pp. 1614 – 1618, 2006.

[8] M. Hamann, M. Krause, and W. Meier, "Lizard – a lightweight stream cipher for power-constrained devices," *IACR Cryptology*, 2016.

[9] *Flip-Flop*, 2009 (accessed February 3, 2009). [Online]. Available: https://www.edgefx.in/digital-electronics-latches-and-flip-flops

[10] *Latches*, 2019. [Online]. Available: https://www.dummies.com/programming/electronics/diy-projects/electronics-basics-what-is-a-latch-circuit/

[11] *Multiplexer-and-De-multiplexer*, 2018. [Online]. Available: https://www.elprocus.com/what-is-multiplexer-and-de-multiplexer-types-and-its-applications

[12] *Finite-state-machine*, 2019. [Online]. Available: http://www.wikiwand.com/en/Finite-state_machine

[13] *State Machine Design*, 2019. [Online]. Available: http://noel.feld.cvut.cz/hw/amd/90005a.pdf

[14] *Clock Divider in VHDL*, 2016. [Online]. Available: https://surf-vhdl.com/how-to-implement-clock-divider-vhdl/

[15] *Area Reduction Strategies*, 2009. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_xst_area_reduction_strategies.htm

[16] *Static timing analysis*, 2019. [Online]. Available: https://www.vlsisystemdesign.com/static-timing-analysis-sta/

[17] *Power Dissipation*, 2008. [Online]. Available: https://ece.uwaterloo.ca/~mhanis/ece637/lecture9.pdf

[18] *Leakage Currents: Sources and Solutions for Low-Power CMOS VLSI*, 2007. [Online]. Available: https://pdfs.semanticscholar.org/6dd4/91311e71f8f2a3454cbc6853f1d68eab9af5.pdf

[19] *pipelining*, 2005. [Online]. Available: https://whatis.techtarget.com/definition/pipelining

[20] B. Vasantha Kumar, "A technique to eliminate glitch power consumption at physical design stage in cmos circuits," *IEEE World Congress on Information and Communication Technologies*, pp. 639–644, 2011.

[21] E. Dubrova and S. S. Mansouri, "An improved implementation of grain," *Cryptography and Security*, vol. arXiv:0910.5595v1, 2009.

[22] S. S. Mansouri and E. Dubrova, "An improved hardware implementation of the grain-128a stream cipher," *Information Security and Cryptology – ICISC*, vol. 15, pp. 278–289, 2012.

[23] M. P. Qing, Wu. and W. Xunwei, "Clock-gating and its application to low power design of sequential circuits," *IEEE Transactions on Circuits and Systems*, vol. 47, pp. 415–420, 2000.

[24] *Counters*, 2019. [Online]. Available: https://www.geeksforgeeks.org/counters-in-digital-logic/

[25] F. Chen-Liang and J. Wen-Ben, "Timing optimization by gate resizing and critical path identification," *lEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 14, pp. 201 – 217, 1995.

[26] S. Banik, V. Mikhalev, and F. Armknecht, "Towards low energy stream ciphers," *IACR Transactions on Symmetric Cryptology*, vol. 2, pp. 1–19, 2018.

[27] A. de Geus, "Logic synthesis speeds asic design," *IEEE Spectrum*, vol. 26, pp. 27 – 31, 1989.

[28] Synopsys, *Design Compiler*. Synopsys, 2005.

[29] ——, *Design Compiler*. Synopsys, 2005, ch. 6.

[30] "Design compiler user guide," vol. F-2011.09-SP2, p. 820, December 2011.

[31] "Design compiler user guide," vol. F-2011.09-SP2, p. 23, December 2011.

[32] H. Bhatnagar, *Advanced ASIC Chip Synthesis Using Synopsys® Design Compiler™ Physical Compiler™ and PrimeTime®*. Kluwer Academic, 2002.

[33] R. W. Keyes, "Moore's law today," *IEEE Circuits and Systems Magazine*, vol. 8, pp. 53 – 54, 2008.

[34] J. M. Rabaey, *Digital Integrated Circuits*. Pearson, 2003.

[35] M. Takahashi, "A 60-mw mpeg4 video codec using clustered voltage scaling with variable supply-voltage scheme," *IEEE J. of Solid-State Circuits*, vol. 33, pp. 1772 – 1780, 1998.

[36] L. Wei, Z. Chen, M. Johnson, and K. Roy, "Design and optimization of dual threshold circuits for low voltage low power applications," *IEEE Trans. on VLSI System*, vol. 16, 1999.

[37] J. Kao, A. Chandrakasan, and D. Antoniadis, "Transistor sizing issues and tool for multi-threshold cmos technology," *Proc. of ACM/IEEE Design Automation Conf.*, vol. 34, pp. 409–414, 1997.

[38] Z. Chen, L. Wei, M. Johnson, and K. Roy, "Estimation of standby leakage power in cmos circuits considering accurate modeling of transistor stacks," *IEEE Int. Conf. on Comput.-Aided Design*, vol. 106, 1998.

[39] T. lnukai and T. Sakura, "Variable threshold voltage cmos (vtcmos) in series connected circuits," *Proceedings. 2001 International Symposium on Low Power Electronics and Design*, 2001.

[40] L. Soongsoo and T. Sakurai, "Run-time voltage hopping for low-power real-time systems," *IEEE/ACM Design Automation Conf.*, vol. 37, pp. 806–809, 2000.

[41] C. Kim and K. Roy, "Dynamic vth scaling scheme for active leakage power reduction, design," *Proc. Design, Automation, and Test in Europe*, pp. 163–167, 2002.

[42] *Introduction to Boolean Algebra*, 2019. [Online]. Available: https://www.allaboutcircuits.com/textbook/digital/chpt-7/introduction-boolean-algebra/

[43] S. M. Kang, *CMOS Digital Integrated Circuits*. McGraw-Hill Education, 2014.

[44] R. Zimmermann and W. Fichtner, "Low-power logic styles: Cmos versus pass-transistor logic," *IEEE Journal of Solid-State Circuits*, vol. 32, pp. 1079 – 1090, 1997.

[45] S. M. Kang, *CMOS Digital Integrated Circuits Analysis & Design*. McGraw-Hill Education, 2014, ch. 7.

[46] *Types of RFID*, 2014. [Online]. Available: https://lowrysolutions.com/blog/what-are-the-different-types-of-rfid-technology

[47] *Secure RFID*, 2019. [Online]. Available: https://www.honeywellaidc.com/products/rfid/tags-labels/it70

[48] H. L. Chae and K. Tymur, "mcrypton – a lightweight block cipher for security of low-cost rfid tags and sensors," *International Workshop on Information Security Applications*, pp. 243–258, 2005.

[49] M. David, D. Ranasinghe, and T. Larsen, "A2u2: A stream cipher for printed electronics rfid tags," *2011 IEEE International Conference on RFID*, 2011.

Appendix A
Code

The VHDL and synthesis code for the implementations can be found at `https://github.com/Noxet/Grain-128AEAD`

Appendix B

Transistor Optimization of the Grain-128AEAD Boolean Expressions

In this appendix some additional transistor optimizations are presented. The Boolean expressions are extracted from the feedback functions and output function of Grain-128AEAD.

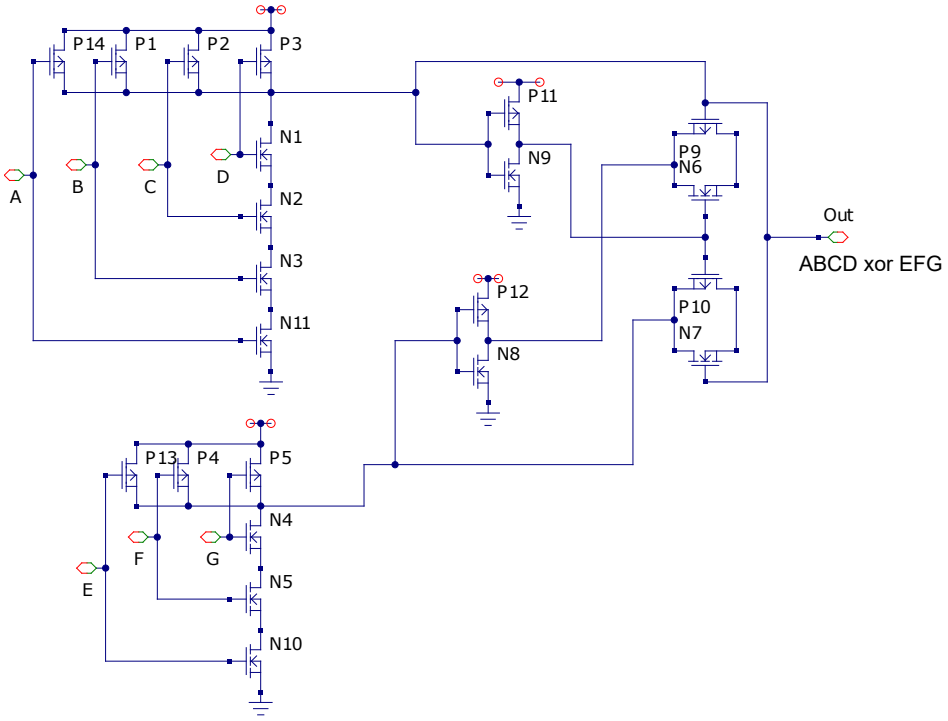


Figure B.1: A figure of a boolean expression "ABCD xor EFG" optimized at transistor level.

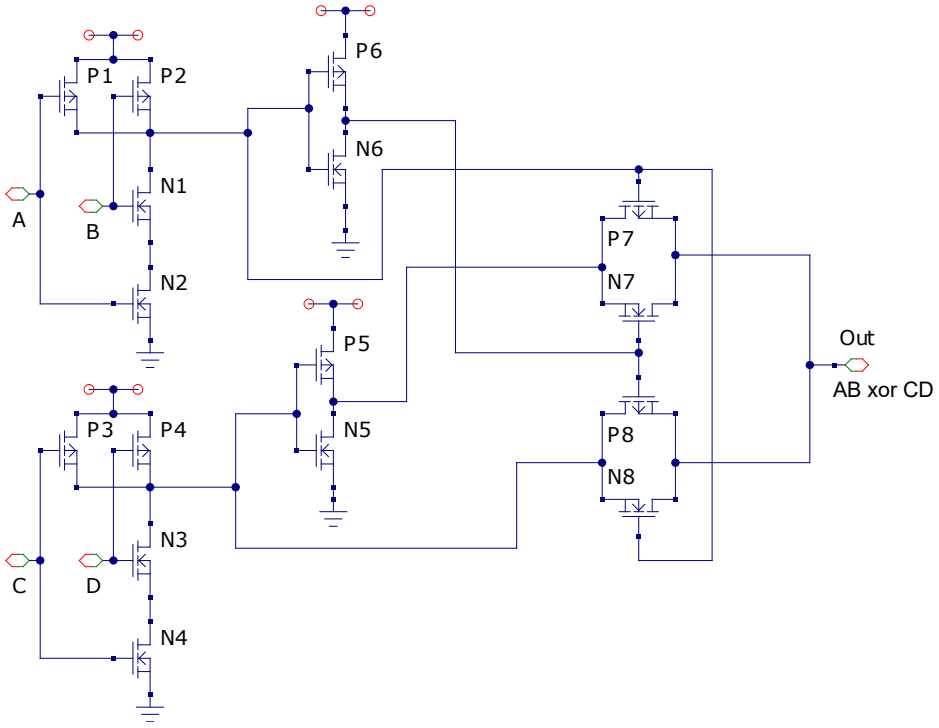


Figure B.2: A figure of a boolean expression "AB xor CD" optimized at transistor level.

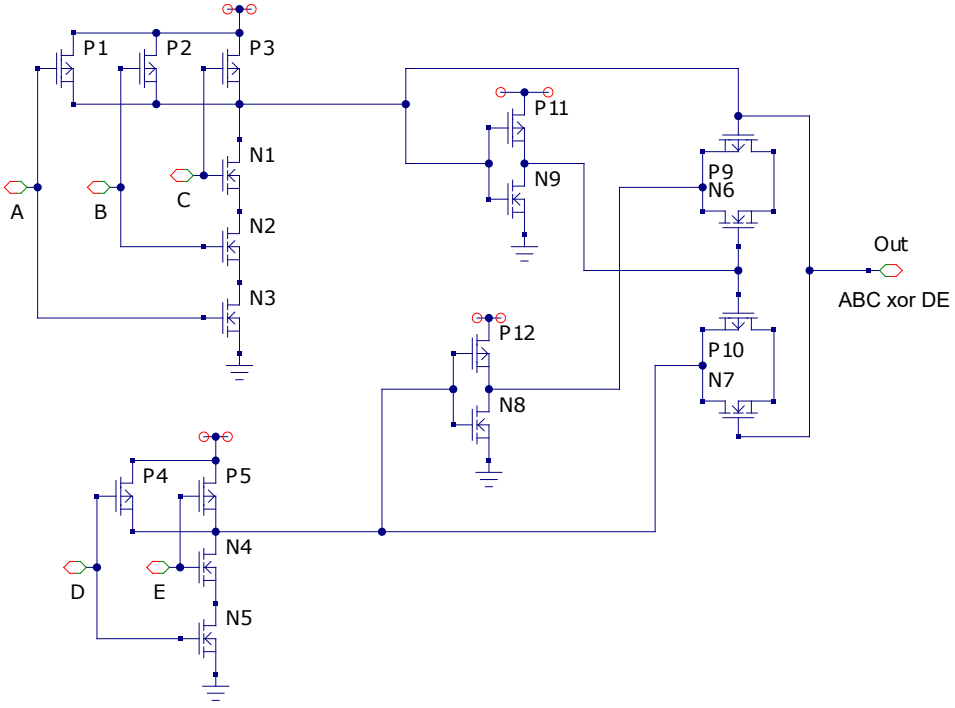


Figure B.3: A figure of a boolean expression "ABC xor DE" optimized at transistor level.

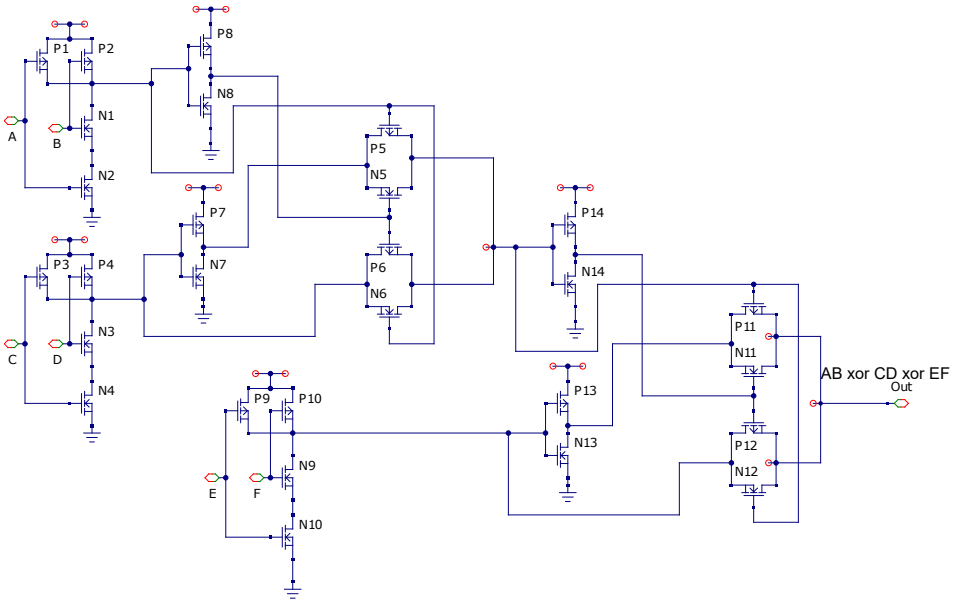


Figure B.4: A figure of a boolean expression "AB xor CD xor EF" optimized at transistor level.