Efficient Hardware Implementations of Grain-128AEAD

Jonathan Sönnerup, Martin Hell, Mattias Sönnerup, and Ripudaman Khattar

Dept. of Electrical and Information Technology, Lund University, Sweden
{jonathan.sonnerup, martin.hell}@eit.lth.se, syntaxnone@gmail.com,
ripudaman11@gmail.com

Abstract. We implement the Grain-128AEAD stream cipher in hardware, using a 65 nm library. By exploring different optimization techniques, both at RTL level but also during synthesis, we first target high throughput, then low power. We reach over 33 Gb/s targeting a high-speed design, at expense of power and area. We also show that, when targeting low power, the design only requires 0.23 μW running at 100 kHz. By unrolling the design, the energy consumed when encrypting a fixed length message decreases, making the 64 parallelized version the most energy efficient implementation, requiring only 11.2 nJ when encrypting a 64 kbit message. At the same time, the best throughput/power ratio is achieved at a parallelization of 4.

Keywords: Grain · stream cipher · ASIC · hardware design · NIST

1 Introduction

Due to the growth and widespread use of resource-constrained connected devices, e.g., in the Internet of Things (IoT), the need for protection against security threats has increased. RFID devices, smart cards, and sensor networks often require low power consumption as they are driven by batteries. The cost of manufacturing an IC chip is correlated to the area. Hence, area efficient designs are needed to reduce the cost when producing large quantities. This puts a demand, not only on the architectural design, but also on the implementation. The implementation may vary largely depending on what techniques are being utilized, both during programming (HDL), but also during synthesis, if aiming for an ASIC. At the same time, high-speed implementations are required for environments with much data, and where low latency is needed.

There are a large number of proposed cryptographic algorithms and several attempts have been made towards identifying suitable algorithms for widespread adoption, e.g., NESSIE, ECRYPT, CRYPTREC and the NIST AES contest. The successful standardization of AES has been followed by more NIST initiatives, most notably the SHA-3 competition, the Post-Quantum Cryptography Standardization Process and the recent Lightweight Cryptography Standardization Process. The latter particularly addresses the need for algorithms that are specifically targeting resource constrained environments [1].

Grain-128AEAD is an instance of the Grain family of stream ciphers. Grain was first proposed in 2005, as an 80-bit stream cipher. The 128-bit variant Grain-128 was presented in 2006 [7], and was successfully cryptanalyzed in [5]. Building upon previous analysis results, a new 128-bit variant with authentication (MAC) support, Grain-128a, was proposed in 2011 [2]. It has also been adopted as an ISO standard [10]. Most recently, Grain-128AEAD supporting Authenticated Encryption with Associated Data, was proposed in 2019 and also submitted to the above mentioned NIST Lightweight Cryptography Standardization Process [8,9]. It is built upon the Grain-128a cipher, but with some added features. There have been several implementations of Grain-128a, most notably the work in [11], where the authors utilize Galois transforms, pipelining and multiple clocks, targeting a high-throughput implementation. While this is important in high-speed applications such as 5G (and beyond) and in servers and gateways which handle multiple connections simultaneously, low energy consumption for certain packet sizes is essential for constrained devices. In [4], the authors target low-energy implementations of stream ciphers, including Grain-128a, discussing multiple techniques for reducing power consumption. In this paper, we discuss several optimization techniques applied to Grain-128AEAD, targeting both high-speed implementations and low-power implementations. Optimizations are considered in both the RTL and at the synthesis level. A small area does not necessarily mean low energy consumption for encrypting a network packet. Adding some area to Grain will reduce the energy for encrypting a packet, even though the power consumption is slightly higher. Our results can be used to better understand the trade-offs between area, power, energy and throughput for the Grain-128AEAD stream cipher. They also provide new benchmark figures for its hardware performance, allowing better and more transparent comparison with other ciphers supporting AEAD. The code is made available at `https://github.com/Grain-128AEAD`.

The paper is outlined as follows. In Section 2, a high-level overview of the Grain-128AEAD design is presented. Section 3 presents a straightforward implementation providing results from where optimization strategies are derived. In Section 4, the utilized RTL optimizations are discussed, whereas in Section 5, different synthesis level optimizations are introduced. Finally, the results are presented in Section 6 and the conclusions are given in Section 7.

2 Grain-128AEAD

This section will provide a brief overview of the Grain-128AEAD design in order to support the optimization approaches discussed later. For a comprehensive design description, we refer to the specification [8,9].

Grain-128AEAD is a cipher in the Grain family. It supports Authenticated Encryption with Associated Data (AEAD) to simultaneously assure confidentiality and authenticity of the data. The overall design is similar to the other ciphers in the family, in particular Grain-128a. It consists of two main building blocks. The first is a pre-output generator consisting of a Linear Feedback Shift Register (LFSR) with feedback function f, a Non-linear Feedback Shift Register

(NFSR) with feedback function g, and a pre-output function denoted h. The pre-output generator outputs a stream y_t. The second block is the authentication block consisting of a shift register and an accumulator. A multiplexer (MUX) is used to control if the pre-output stream y_t is used for authentication, z'_i, or for keystream, z_i. The architectural overview of Grain-128AEAD is depicted in Fig. 1.

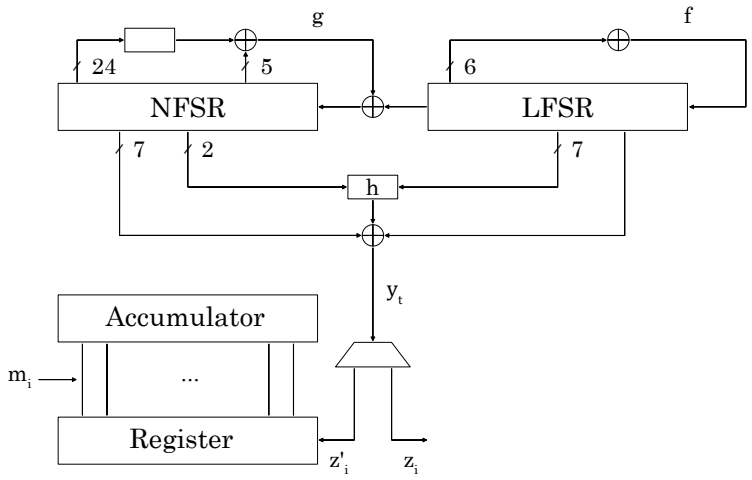

Fig. 1. An architectural overview of Grain-128AEAD.

2.1 Phases of Grain-128AEAD

For the hardware implementation, we logically divide the cipher into three phases. The first phase is the *loading phase*, in which the shift registers are loaded with the key and the nonce. Next, Grain-128AEAD enters the *initialization phase* in which the registers and the authentication module are initialized. Finally, the cipher enters the *running phase*, in which pre-output is generated both for encryption and authentication.

2.2 Pre-output Generation

The pre-output generator uses a 128-bit LFSR and a 128-bit NFSR. The content, at instance t, of the LFSR is denoted as $S_t = [s_0^t, s_1^t, \ldots, s_{127}^t]$, and similarly for the NFSR, $B_t = [b_0^t, b_1^t, \ldots, b_{127}^t]$. Together, the two FSRs form the 256-bit state of the generator. The feedback polynomial of the LFSR, f, may be written as the recurrence relation given by

$$\begin{aligned} s_{127}^{t+1} &= s_0^t + s_7^t + s_{38}^t + s_{70}^t + s_{81}^t + s_{96}^t \\ &= \mathcal{L}(S_t). \end{aligned}$$

The feedback polynomial of the NFSR, g, may be written as the recurrence relation given by

$$\begin{aligned} b_{127}^{t+1} &= s_0^t + b_0^t + b_{26}^t + b_{56}^t + b_{91}^t + b_{96}^t + b_3^t b_{67}^t + b_{11}^t b_{13}^t \\ &\quad + b_{17}^t b_{18}^t + b_{27}^t b_{59}^t + b_{40}^t b_{48}^t + b_{61}^t b_{65}^t + b_{68}^t b_{84}^t \\ &\quad + b_{22}^t b_{24}^t b_{25}^t + b_{70}^t b_{78}^t b_{82}^t + b_{88}^t b_{92}^t b_{93}^t b_{95}^t \\ &= s_0^t + \mathcal{F}(B_t). \end{aligned}$$

The Boolean function h_t uses bits from both the LFSR and the NFSR, and is defined as

$$h_t = b_{12}^t s_8^t + s_{13}^t s_{20}^t + b_{95}^t s_{42}^t + s_{60}^t s_{79}^t + b_{12}^t b_{95}^t s_{94}^t.$$

The output, y_t, from the pre-output generator is given by

$$y_t = h_t + s_{93}^t + \sum_{j \in \mathcal{A}} b_j^t,$$

where $\mathcal{A} = \{2, 15, 36, 45, 64, 73, 89\}$.

After the initialization phase, the pre-output is used to generate keystream bits z_i for encryption and authentication bits z_i' to update the register in the accumulator generator. The keystream is generated as

$$z_i = y_{384+2i},$$

i.e., every even bit (counting from 0) from the pre-output generator is taken as a keystream bit. The authentication bits are generated as

$$z_i' = y_{384+2i+1},$$

i.e., every odd bit from the pre-output generator is taken as an authentication bit.

2.3 Authentication Module

The authenticator generator consists of a 64-bit shift register and a 64-bit accumulator. The content of the shift register, at instance i, is denoted $R_i = [r_0^i, r_1^i, \ldots, r_{63}^i]$, and similarly for the accumulator, the content is denoted $A_i = [a_0^i, a_1^i, \ldots, a_{63}^i]$. The accumulator is updated as

$$a_j^{i+1} = a_j^i + m_i r_j^i, \qquad 0 \le j \le 63, \quad 0 \le i \le L, \tag{1}$$

where m_i is the ith message bit, and the shift register is updated as

$$\begin{aligned} r_{63}^{i+1} &= z_i', \\ r_j^{i+1} &= r_{j+1}^i, \qquad 0 \le j \le 62. \end{aligned}$$

2.4 Loading and Initialization

After reset, the cipher must be loaded and initialized. The loading is performed as follows. Let k_i be the key bits where $0 \leq i \leq 127$, and let IV_i be the nonce (IV) bits where $0 \leq i \leq 95$. The NFSR is loaded with the key, i.e., $b_i^0 = k_i,\ 0 \leq i \leq 127$. The first 96 bits of the LFSR is loaded with the nonce, i.e., $s_i^0 = IV_i,\ 0 \leq i \leq 95$, and the last 32 bits are filled with 31 ones and a zero, i.e., $s_i^0 = 1,\ 96 \leq i \leq 126,\ s_{127}^0 = 0$. Next, in the initialization phase, the cipher is clocked 256 times, feeding back the pre-output adding it with the input to the NFSR and LFSR, using the XOR operation, i.e.,

$$s_{127}^{t+1} = \mathcal{L}(S_t) + y_t, \quad 0 \leq t \leq 255,$$
$$b_{127}^{t+1} = s_0^t + \mathcal{F}(B_t) + y_t, \quad 0 \leq t \leq 255.$$

Next, the shift register and accumulator in the authenticator are initialized with the pre-output stream as

$$a_j^0 = y_{256+j}, \quad 0 \leq j \leq 63,$$
$$r_j^0 = y_{320+j}, \quad 0 \leq j \leq 63.$$

At the same time, the key is added to the feedback of the LFSR as

$$s_{127}^{t+1} = \mathcal{L}(S_t) + k_{t-256}, \quad 256 \leq t \leq 383,$$

while the NFSR is updated as

$$b_{127}^{t+1} = s_0^t + \mathcal{F}(B_t), \quad 256 \leq t \leq 383.$$

The loading phase and the initialization phase are summarized in Fig. 2.

3 A Straightforward Approach

The stream cipher is implemented in hardware using RTL design in VHDL. For synthesis and power simulation, the Synopsys Design Compiler 2013.12 is used along with a 65 nm library from ST Microelectronics, `stm065v536`. The number of required gates, and the number of transistors in a gate depends on the library used and may vary by a large degree. In this paper, the area of the designs are given in gate equivalents (GE), which is the physical area divided by the area of a 2-input NAND gate for the given library.

A straightforward approach is taken when implementing the cipher, closely following the proposed architectural design in [8,9] - the FSRs are in Fibonacci configuration parallelized at most 32 times. The key and nonce are simultaneously loaded serially, and the accumulator is loaded by first loading the shift register, then moving the values to the accumulator. For the parallelized implementations, the loading phase is sped up by a factor n, where n is the parallelization level. A simple Finite State Machine (FSM) is used to keep track of

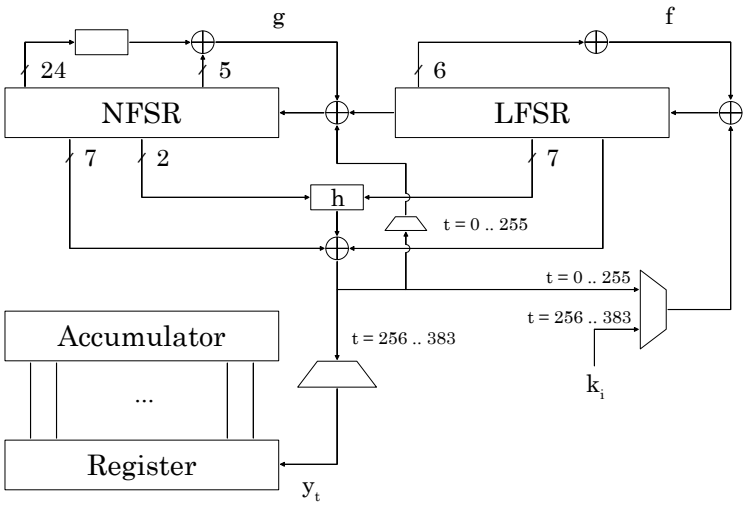

Fig. 2. An architectural overview of the initialization in Grain-128AEAD.

the different phases, or states, in order to control the data paths. Finally, we let the synthesizing tool optimize for speed. This implementation and synthesis is used for benchmarking and comparison with our optimized implementations.

In order to improve the bottlenecks in the synthesized design, we must analyze the critical paths. Similar to [11], we define the following delays:

- D_n: the maximal delay from any NFSR or LFSR flip-flop to any other NFSR or LFSR flip-flop.
- D_y: the maximal delay from any NFSR or LFSR flip-flop to the output, via the y function.
- D_{ya}: the maximal delay from any NFSR or LFSR flip-flop to any accumulator flip-flip, via the y function.
- D_a: the maximal delay from any flip-flop in the authentication section to any accumulator flip-flop, or output.
- D_{yn}: the maximal delay from a flip-flop of the NFSR or LFSR through the y function to the first flip-flop of the NFSR. This path only exists during initialization of the cipher, via a MUX.

The critical paths are highlighted in Fig. 3. Note that y_{out}, after initialization, corresponds to z as in Fig. 1. Similarly, y_{accum}, after initialization, corresponds to z'.

Synthesizing the design yields the results shown in Table 1, where the propagation delay of the critical path is listed. D_{yn} is only available during initialization. In the running state of the cipher, it is instead D_n which is the critical path. These critical delays, together with D_{ya}, will be targeted in the next section.

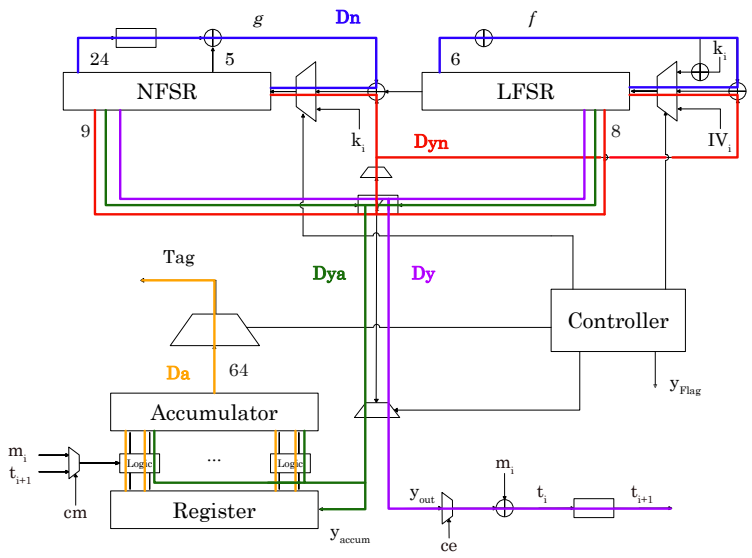

Fig. 3. Architectural overview of Grain-128AEAD with the following potential critical paths highlighted: D_n (blue), D_y (purple), D_{ya} (green), D_a (yellow), and D_{yn} (red).

Table 1. Clock periods and critical paths of the straightforward implementation, for different levels of parallelization.

	x1	x2	x4	x8	x16	x32
Period (ns)	490	610	640	690	770	840
Critical Path	D_{yn}	D_{yn}	D_{yn}	D_{yn}	D_{ya}	D_{ya}

4 RTL Level Optimizations

Here, we present the architectural optimization techniques utilized when targeting speed, area, and power. In particular, for speed improvements, we aim to lower the delay induced by the critical paths in Table 1. We start by presenting a general optimization technique, Galois transform, to reduce the D_n path. Then, we utilize a similar technique in order to reduce the path D_{yn}, discussed in Section 4.2.

4.1 Galois Transformation

As described earlier, the D_n path lies between two flip-flops (any flip-flop to the right most flip-flop in Fig. 1) in the FSRs, for the 1, 2, 4, and 8 parallelized versions, causing a bottleneck in the running mode. The usual strategy would be to pipeline the FSRs by inserting flip-flops at well chosen positions. In order to pipeline a design, the delay elements can only be inserted in the feed-forward cutset of the corresponding graph [12]. This is not possible for the FSRs due to

their intrinsic feedback property. In order to decrease the propagation delay in the D_n path, the cipher may be transformed from its normal Fibonacci configuration to a Galois configuration. In the Fibonacci configuration, the flip-flops are updated with the value of the previous flip-flop every clock cycle, i.e., $x_i = x_{i+1}$, except for x_{n-1} which gets updated with the result of the feedback polynomial, i.e., $x_{n-1} = f(x)$. In the Galois configuration, some of the flip-flops get updated with the result of a function of other flip-flops, i.e., $x_i = g(x)$. The Galois configuration leads to shorter propagation delays due to the feedback function in Fibonacci being split up and put in between flip-flops.

For an LFSR, the Fibonacci to Galois transform is a one-to-one mapping. For an NFSR, multiple Galois configurations exist for a given Fibonacci configuration [6]. The Galois transform is identical to Grain-128a, hence we refer to [11] for details. Grain-128a can not be transformed to a Galois configuration for a parallelization level above 16. The same holds for Grain-128AEAD.

4.2 Transforming the y Function

During the initialization phase, the y function is being fed back to the shift registers, forming an FSR. As in the previous section, it is not possible to insert pipelines due to the lack of a feed-forward cutset. Instead, similar to the Galois transform, it is possible to transform the y function in such way that it is split up and fed back to different registers, which reduces the critical path, D_{yn}. The transformed functions are denoted as $Y_{125}, Y_{126},$ and Y_{127}. For a parallelization level of 8 and 16, only Y_{126} and Y_{127} may be used. As an example for the non-parallelized version, the functions are given by

$$\begin{aligned} Y_{127} &= b_{12}s_8 + s_{13}s_{20} + b_{95}s_{42}, \\ Y_{126} &= b_{11}b_{94}s_{93} + b_{72} + b_1 + s_{50}s_{78}, \\ Y_{125} &= s_{91} + b_{87} + b_{13} + b_{34} + b_{43} + b_{62}. \end{aligned}$$

After initialization, during the running state, the feedback loop is disconnected. This allows for insertion of pipelines steps. By combining both Galois-like transformation and pipelining, both D_{yn} and D_y may be reduced, see Fig. 4. The controller switches between the two methods when required.

4.3 Isolating the Authentication Module

The accumulator is updated using the values in the shift register as in Eq. (1). When parallelizing the design, the accumulator is updated with the corresponding shift register plus values from the shift registers with larger index as

$$a_j^{i+1} = a_j^i + \sum_{k=0}^{\frac{p}{2}-1} m_{i+k} \cdot r_{j+k}^i, \qquad p \geq 4,$$

where p is the parallelization level. For $p = 2$, the update expression is equivalent to Eq. 1, since 1 bit is generated every clock cycle for authentication. Note that

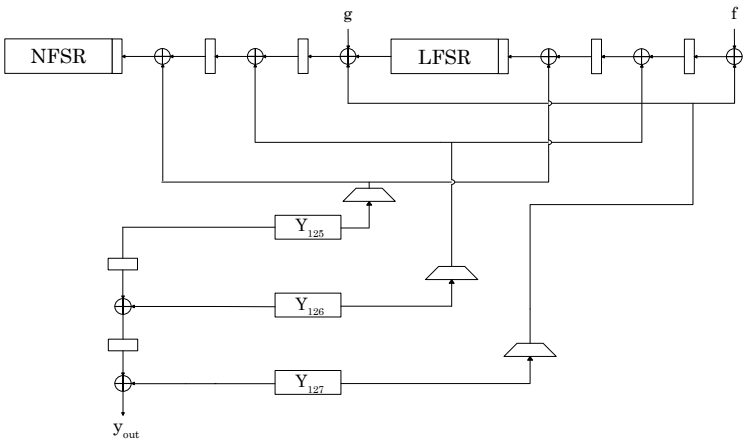

Fig. 4. The transformation of the y function along with pipeline steps and control logic.

for a parallelization level of 4 and above, some values have not yet been shifted in to the shift register, e.g., for $p = 4$, the register a_{63} is updated as

$$a_{63}^{i+1} = a_{63}^{i} + \left(m_i \cdot r_{63}^{i}\right) + \left(m_{i+1} \cdot r_{64}^{i}\right),$$

where r_{64} has been generated but is not located in the shift register. This can be seen as a future value, and requires extra combinational logic to handle. This means that the path D_{ya} becomes longer and, for the higher levels of parallelization, affects the timing of the design, as seen in Table 1.

In order to make the D_{ya} path shorter, a pipeline step is inserted between the y function and the accumulator logic, as shown in Fig. 5. This allows for the throughput to increase due to a higher clock frequency, but adds a 1 clock cycle delay to the accumulator calculation. Note that this does not affect the security.

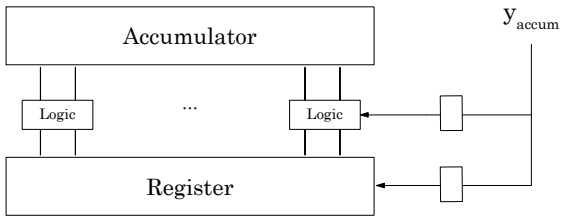

Fig. 5. Isolating the authentication module using pipelining.

4.4 Optimizing the Controller

A controller is a unit responsible for managing the data flow and operations, such as the feedback loop, when to accumulate data, and when to encrypt the plaintext.

The straightforward implementation of the controller is a finite state machine (FSM) with states corresponding to loading, initialization, and the running phase. The controller can also be implemented using a LFSR with some combinational logic. Experiments with LFSRs gave roughly the same results as the FSM generated by the synthesizer. Here, we explore an alternative strategy to keep track of the state by using a clock divider and a shift register, which often, but not always, gave better performance.

The idea is to start from an empty shift register (all zeroes), and shifting in a 1 each clock cycle. This means that after n clock cycles, there is a 1 at index n in the shift register. We can use this to control the logic in the cipher via, e.g., MUXes. However, Grain requires 512 clock cycles before it produces the first bit, i.e., 128 for loading key and nonce plus 384 for the initialization rounds. This results in a shift register with 512 flip-flops, which is not desirable due to the huge increase in size. Instead, note that the resolution given by 512 registers is not fully utilized, since we ignore most of the intermediate values, hence we can reduce the size. With a reduced size, we must compensate with a lower clock frequency for controlling the design at the correct time instances. This is done using a clock divider to slow the shift register down by a factor 2^k. The value of $2^k p$ can not exceed 128, since the least amount of clock cycles that we need to keep track of are 128, for loading key and nonce. The number of registers required depends on the level of parallelization, p, and the k value as $512/(2^k p)$. Taking the clock divider registers into account gives an expression for the total number of registers required as

$$\frac{512}{2^k p} + k.$$

This design does not require much hardware, e.g., letting $p = 16, k = 3$ results in 7 flip-flops. In this paper, the largest value of k is selected, for every level of parallelization, i.e., $k = \log_2(128/p)$.

The index of the shift register corresponding to the different phases are calculated as

$$i_{\text{load}} = \frac{128}{kp}, \quad i_{\text{init}} = \frac{128 + 256}{kp}, \quad i_{\text{run}} = \frac{512}{kp}.$$

For the control MUXes that remain in their state after being activated may be directly connected to the controller. For the control MUXes that are only activated during a single state, an inverter together with an AND-gate is required.

4.5 Unrolling

Grain natively supports parallelization up to 32 times by using multiple feedback and output functions, f, g and y. However, there is nothing preventing us to go

further, at RTL level. Consider AES, where multiple rounds of the AES round function is executed one after the other, in a loop structure. The block is fed back via MUXes to realize successive iterations. Unrolling the AES rounds to a level L means that we put L AES round functions serially in a single combinational block as done in [13].

When unrolling Grain, we do not utilize multiple instances of the FSRs, but rather the feedback functions. When reaching a parallelization level above 32, the feedback functions start to interact. Revising the feedback function f of the LFSR, we can extend this to include multiple copies of f, denoted as f_k. The index k is related to the level of parallelization, where the highest k value plus 1 ($\max(k) + 1$) equals the parallelization level. Expressing the bits as a sequence, we can write the expression as

$$s_{i+128+k} = s_{i+0+k} + s_{i+7+k} + s_{i+38+k} + s_{i+70+k} + s_{i+81+k} + s_{i+96+k}.$$

With $k = 31$, the bit with the highest index in f_{31} is $s_{i+96+31} = s_{i+127}$, by design. However, $k = 32$ includes bit index s_{i+128} in f_{32}, which is not a register index, but rather the output from the function f_0. Thus, the two feedback functions are connected, as seen in Fig. 6. Increasing k leads to more interconnection between the feedback functions, thus increasing the propagation delay. Parallelization of higher degrees for the authentication module continues the approach described in Section 4.3.

Apart from an increase in throughput, unrolling also allows for energy saving as more data is processed in a single clock cycle which reduces the total switching activity and the number of clock cycles it takes to complete a computation [3].

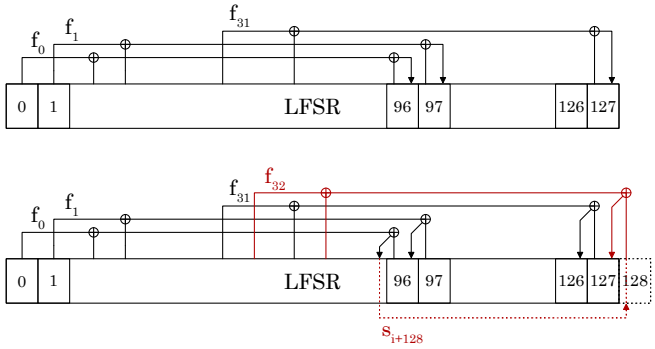

Fig. 6. Example of unrolling above the specified level of 32. The bottom picture shows the structure when using a parallelization level of 33. The last feedback function, f_{32}, needs to read the value from a flip flop that does not exist, the register index 128. Instead, this value is the output from f_0, which would have been stored in register index 128 if it existed. We can therefore take the output from f_0 as an input to f_{32}. From this it is clear that the propagation delay increases when exceeding the specified level of parallelization.

5 Synthesis Level Optimization

Synthesis is the process where high-level RTL code, like VHDL and Verilog, is used to generate a gate-level netlist. There are 3 steps involved during synthesis:

1. Translation: The RTL code is converted to a technology-independent representation of Boolean expressions.
2. Optimization: The Boolean expressions are minimized, with respect to gates, using a minimization algorithm.
3. Technology mapping: The Boolean expressions are mapped to a library, based on the used technology, in order to produce a gate-level netlist.

Design Vision requires the RTL code, design constraints, and a standard cell library, in order to generate a netlist. Design Vision offers two commands used for compiling - `compile` and `compile_ultra`. The `compile_ultra` command is used for designs with tight timing constraints, and produces better quality of results compared to `compile`. Hence, only `compile_ultra` is used during synthesis.

There are several compiler options to be utilized during synthesis. Here, we highlight some of the most commonly used features:

- **Structuring** - The process where intermediate variables are added to the design, in order to reduce area. The synthesis tool factors out common sub functions that mostly reduces the area and turn them into intermediate variables.
- **Flattening** - Here, the tool converts combinational logic paths into a sum-of-products representation. This often leads to a faster design due to the combinational logic requiring only two levels. Consequently, it may lead to an increase in area.
- **Ungrouping** - A common strategy when implementing a design is to group different parts of the code, to have a hierarchical design. This leads to well structured design and it is easy to analyze the synthesized design. By ungrouping, the tool is less constrained and may reorganize the design as it see fits, which may lead to a faster design, at the expense of area.
- **Clock Gating** - Insert control logic in order to regulate the clock signal, either to shut it down at time instances, or to modify the clock pulse. This may be used to save energy.

5.1 Transistor Types

In a complementary MOS (CMOS) design, both NMOS and PMOS are used. When one is conducting, the other is not, resulting in very small static power consumption, given by

$$P_s = V_{dd} \cdot I_{\text{leakage}},$$

where V_{dd} is the supply voltage. The leakage current depends on the threshold voltage, V_{th}, and a transistor with low V_{th}, LVT, has higher leakage current than a transistor with a high V_{th}, HVT. To minimize leakage current, HVT

transistors are most suitable for power efficient implementations. For the high-speed implementations, LVT transistors are most suitable since they allow to increase the switching speed.

6 Synthesis Results

Providing results for all possible combinations of implementations, synthesis options and transistors would become very verbose. Instead, to facilitate a more clear and concise presentation and basis for comparison, the implementations considered will be as follows.

- **Straightforward implementation**. This implementation will closely follows the architectural design, using no optimization techniques. The design is synthesized for high speed utilizing LVT transistors, and different synthesis flags to achieve the best result.
- **High speed implementation**. Here, we apply all viable optimizations at RTL and synthesis level and synthesize for high speed using LVT transistors.
- **Low Power implementation**. In the low power scenario, we cut back the clock frequency, employing only unrolling and the improved controller as optimization techniques. Both the LVT and HVT transistors are used for comparison of power consumption.

6.1 Straightforward Implementation

Results for the straightforward implementation are shown in Table 2, using no RTL optimizations, but synthesized for maximum speed. Synthesis options such as flattening, structuring, and ungrouping were utilized. Neither flattening nor structuring affected the result significantly. Only the grouping/ungrouping option made a difference. This difference was typically in the order of 0.02 ns for the period. In the result tables, the best result is presented, and we also highlight whether grouping (G) or ungrouping (U) yielded the result.

Similar to [4], we also calculate the energy consumed when encrypting 1 block of data (64 bits) and 1000 blocks, shown in Table 3. For example, encrypting 1 block of data, in the non-parallelized ($n = 1$) version at 2.04 GHz, requires 128 (loading key and IV) + 384 (initialization) + 128 (64 keystream bits + 64 bits for authentication) = 640 clock cycles. The energy consumed results in 640×0.49 ns $\times 170$ μW $= 0.053$ nJ. Note that the number of clock cycles required for encryption is inversely proportional to n.

The non-parallelized version has the highest clock frequency, but the lowest throughput (thrp). The clock period does not scale at the same rate as n, which allows the higher levels of parallelization to have higher throughput. Between $n = 1$ and $n = 32$, the throughput increases by a factor 19, whereas the area only increases by a factor 3.8. For $n = 32$, we achieve the highest throughput to area ratio. For $n = 4$, the highest throughput to power is reached along with the lowest energy consumption, making it the most power efficient version.

Table 2. Straightforward implementation synthesized for high speed. The throughput per area is given in kbit/s per GE. The throughput per power is given in Gb/s per mW. The synthesis optimization (Opt.) shows whether grouping (G) or ungrouping (U) gave the best result.

n	Period (ns)	Freq. (GHz)	Thrp. (Gb/s)	Area (GE)	Power (mW)	Thrp. / Area	Thrp. / Power	Opt.
1	0.49	2.04	1.02	2689	0.17	182	5.99	U
2	0.61	1.64	1.64	2776	0.14	284	11.76	G
4	0.64	1.56	3.12	3333	0.21	450	14.93	G
8	0.69	1.44	5.76	4324	0.42	640	13.70	G
16	0.77	1.29	10.32	6265	0.92	792	11.24	G
32	0.84	1.19	19.04	10226	2.54	895	7.52	G

Table 3. This shows the energy consumption for the straightforward implementation, processing 1 and 1000 blocks of data. 1 block equals 64 bits of data.

Energy (nJ)	x1	x2	x4	x8	x16	x32
1 Block	0.053	0.027	0.022	0.023	0.028	0.042
1000 Blocks	10.70	5.48	4.31	4.65	5.69	8.57

The ungrouping option seems to be worse for all versions except $n = 1$. Using the ungrouping option led to a higher clock frequency, but the tool reported fanout violations which it could not resolve. Choosing to only consider results without any violations, these results were omitted.

6.2 High Speed Implementation

Here, we apply the techniques described earlier in order to increase the throughput of the design. For the parallelization levels 1, 2, 4, 8 and 16, Galois transform together with y transform, isolation of authentication module, and the optimized controller are utilized. For the 32 (parallelized) and 64 (unrolled) versions, only isolation of authentication and the optimized controller are possible. Transformation of the y function is not applicable due to similar constraints as for the Galois transformation of the shift registers.

The results for the optimized implementation are presented in Table 4. The energy consumption for a given message length is given in Table 5, where the highest speed at each level of parallelization from Table 4 is used. Table 4 shows an increase in throughput for every level of parallelization, at the expense of increased power consumption. However what is interesting is that the optimized controller actually reduces the power consumption while increasing the throughput for $n = 32$ and $n = 64$. For $n = 32$, the power consumption is lower than the straightforward implementation, while for $n = 64$, it is just 0.22 mW more

Table 4. Results for the high-speed implementation, with optimized controller on greyed background and regular controller on white. The throughput per area is given in kbit/s per GE. The throughput per power is given in Gb/s per mW.

n	Period (ns)	Freq. (GHz)	Thrp. (Gb/s)	Area (GE)	Power (mW)	Thrp. / Area	Thrp. / Power	Opt.
1	0.43	2.3	1.15	2791	0.24	412	4.79	U
	0.40	2.5	1.25	2645	0.25	472	5.00	U
2	0.46	2.17	2.17	2800	0.21	776	10.33	G
	0.43	2.32	2.32	2695	0.23	861	10.09	G
4	0.47	2.13	4.26	3335	0.29	1277	14.69	G
	0.48	2.08	4.16	3199	0.29	1300	14.34	U
8	0.48	2.08	8.32	4537	0.67	1834	12.42	G
	0.46	2.17	8.68	4448	0.67	1951	12.96	G
16	0.50	2.00	16.00	6270	1.44	2552	11.11	G
	0.48	2.08	16.64	7118	1.55	2338	10.74	U
32	0.69	1.45	23.20	9148	2.66	2536	8.72	G
	0.64	1.56	24.96	9206	1.78	2710	14.02	U
64	1.00	1.00	32.00	16618	4.76	1926	6.72	G
	0.95	1.05	33.60	16958	2.76	1982	12.17	U

Table 5. This shows the energy consumption for the high-speed implementation, processing 1 and 1000 blocks of data. 1 block equals 64 bits of data.

Energy (nJ)	x1	x2	x4	x8	x16	x32	x64
1 Block	0.064	0.032	0.022	0.025	0.030	0.023	0.026
1000 Blocks	12.85	6.35	4.38	4.95	5.98	4.58	5.26

than the straightforward, 32 parallelized version, but with a 76% increase in throughput. We can again note that a parallelization level of 4 yields the highest throughput per power along with the lowest energy consumption. The optimized controller also affects the throughput per power the most for $n = 32$ and $n = 64$.

As also seen in Table 4, ungrouping the design led to a higher throughput when using the optimized controller.

It is clear that the area increases with higher throughput, due to higher levels of parallelization. An important metric is the throughput per area, which measures area efficiency. From the table, we find that the most area efficient implementation occurs when $n = 32$, using the improved controller. This is not surprising since increasing parallelization should only require a "small" increase in area, by design. This is an important feature in the Grain family of stream ciphers.

6.3 Low Power Implementation

When targeting low power, a clock period must be specified. Many low-power devices run at frequencies around 10 MHz. The ISO standard for contactless smart cards, ISO/IEC 15693, defines the frequency to be 13.56 MHz. Older proximity cards operate at 125 kHz. Thus, for low power applications, we choose to synthesize the design at the clock frequencies 100 KHz and 10 MHz, shown in Table 6 and 7, respectively.

The synthesis script utilizes `compile_ultra` with clock gating and low power transistors (HVT). For comparison, we also synthesize the design using the high speed scripts and select the best result, for comparison. The RTL optimization implemented for low power is unrolling along with the optimized controller.

Table 6. Result for the low power implementation running at 100 kHz. Here, we compare the speed script (S_s), the power (P_s) script, and the power script using the optimized controller (P_{opt}). 1 block equals 64 bits of data.

n	**Area** (GE)			**Power** (μW)			**Energy** (nJ)	
	S_s	P_s	P_{opt}	S_s	P_s	P_{opt}	1 block	1000 blocks
1	2509	2375	2337	2.29	0.23	0.26	1.47	296
	-	-5%	-7%	-	-89%	-88%	-	-
2	2592	2588	2511	2.33	0.28	0.30	0.90	180
	-	0%	-3%	-	-87%	-86%	-	-
4	2952	2950	2862	2.33	0.29	0.32	0.46	93.2
	-	0%	-3%	-	-87%	-86%	-	-
8	3695	3692	3594	2.76	0.31	0.35	0.25	49.8
	-	0%	-2%	-	-88%	-87%	-	-
16	5168	5158	5053	3.77	0.42	0.39	0.16	31.3
	-	0%	-2%	-	-89%	-90%	-	-
32	8168	8126	7950	5.93	0.62	0.46	0.09	18.5
	-	0%	-3%	-	-90%	-92%	-	-
64	14100	14093	13800	10.89	1.08	0.63	0.06	12.7
	-	0%	-2%	-	-90%	-94%	-	-

Overall, there was very little difference in area when synthesizing for high speed and low power using the standard controller. For such low frequencies, the timing is easily met and the tool optimizes for area in both cases, thus there is not much to improve. For the power however, there is a clear difference using HVT transistors compared to LVT. There is a 86 - 92% reduction in power consumption for all levels of parallelization running at 100 kHz, and a 19 - 37% power reduction for 10 MHz. In the design paper of Grain [8,9], the authors used HVT transistors when synthesizing for high speed. This led to a lower clock frequency and a higher power consumption than the figures in Table 4.

Table 7. Result for the low power implementation running at 10 MHz. Here, we compare the speed script (S_s), the power (P_s) script, and the power script using the optimized controller (P_{opt}). 1 block equals 64 bits of data.

n	**Area** (μm^2)			**Power** (μW)			**Energy** (nJ)	
	S_s	P_s	P_{opt}	S_s	P_s	P_{opt}	1 block	1000 blocks
1	2510	2375	2337	33.66	22.07	25.21	1.41	283
	-	-5%	-6%	-	-34%	-25%	-	-
2	2592	2589	2511	33.96	26.93	29.13	0.86	173
	-	0%	-3%	-	-21%	-14%	-	-
4	2952	2951	2862	34.43	27.38	31.05	0.44	88.0
	-	0%	-3%	-	-20%	-10%	-	-
8	3695	3693	3595	36.83	29.38	33.59	0.24	47.2
	-	0%	-3%	-	-20%	-9%	-	-
16	5168	5162	5057	44.02	39.49	36.93	0.15	29.7
	-	0%	-2%	-	-10%	-16%	-	-
32	8172	8128	7951	66.02	57.08	41.66	0.08	16.7
	-	0%	-2%	-	-13%	-37%	-	-
64	14101	14093	13810	117.4	97.39	55.93	0.06	11.2
	-	0%	-2%	-	-17%	-52%	-	-

Hence, HVT should only be used for lower frequencies where power is the main concern, whereas LVT should be used for higher frequencies where the target is speed.

Even though the power consumption increases with increasing n, the energy cost decreases since the computation can be done in much shorter time. This leads to the unrolled 64-parallelized version being the most energy efficient implementation for a given message length.

The optimized controller reduces the area in all cases at expense of higher power consumption for $n = 1, 2, 4, 8$. For $n = 16, 32, 64$, the power consumption is reduced when using the optimized controller.

7 Conclusions

In this paper, we implemented Grain-128AEAD and investigated the impact of different implementation strategies, from RTL to synthesis-level design, to either achieve high throughput or low power consumption.

By utilizing different optimization techniques, we reduced the power by up to 94% compared to a straightforward implementation. By unrolling the design, the power consumption increases while the energy for encrypting a message of fixed size decreases. The 64-level parallelization implementation requires only 11.2 nJ when encrypting 64 kbits of data compared to 283 nJ for the non-parallelized

version. For the high-speed implementation, the maximum throughput reached is 33.6 Gb/s. It is not obvious in which cases the (un)grouping option yields the best result, hence both options should be analyzed in order to find the best result. We notice that a parallelization level of 4 yields the most power efficient implementation, both for the straightforward implementation and the high-speed one. The experiments show that Grain is well suited both in high-speed applications as well as on constrained devices requiring low power consumption.

8 Source Code

The source code can be found at
`https://drive.google.com/open?id=14NYrM9yyV1MP6UM2IHMJmWEybOcLXHU8`.

Acknowledgements This paper was supported by the Swedish Foundation for Strategic Research, grant RIT17-0032.

References

1. National institute of standards and technology: Proposed submission requirements and evaluation criteria for the post-quantum cryptography standardization process (2018), `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf`
2. Ågren, M., Hell, M., Johansson, T., Meier, W.: Grain-128 a: a new version of Grain-128 with optional authentication. International Journal of Wireless and Mobile Computing **5**(1), 48–59 (2011)
3. Banik, S., Bogdanov, A., Regazzoni, F.: Exploring energy efficiency of lightweight block ciphers. In: Dunkelman, O., Keliher, L. (eds.) Selected Areas in Cryptography – SAC 2015. pp. 178–194. Springer International Publishing, Cham (2016)
4. Banik, S., Mikhalev, V., Armknecht, F., Isobe, T., Meier, W., Bogdanov, A., Watanabe, Y., Regazzoni, F.: Towards low energy stream ciphers. IACR Transactions on Symmetric Cryptology **2018**(2), 1–19 (Jun 2018). https://doi.org/10.13154/tosc.v2018.i2.1-19, `https://tosc.iacr.org/index.php/ToSC/article/view/886`
5. Dinur, I., Shamir, A.: Breaking Grain-128 with dynamic cube attacks. In: Joux, A. (ed.) Fast Software Encryption. pp. 167–187. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
6. Dubrova, E.: A transformation from the Fibonacci to the Galois NLFSRs. IEEE Transactions on Information Theory **55**(11), 5263–5271 (Nov 2009). https://doi.org/10.1109/TIT.2009.2030467
7. Hell, M., Johansson, T., Maximov, A., Meier, W.: A stream cipher proposal: Grain-128. In: 2006 IEEE International Symposium on Information Theory. pp. 1614–1618 (July 2006). https://doi.org/10.1109/ISIT.2006.261549
8. Hell, M., Johansson, T., Meier, W., Sönnerup, J., Yoshida, H.: An AEAD variant of the Grain stream cipher. In: Carlet, C., Guilley, S., Nitaj, A., Souidi, E.M. (eds.) Codes, Cryptology and Information Security. pp. 55–71. Springer International Publishing, Cham (2019)

9. Hell, M., Johansson, T., Meier, W., Sönnerup, J., Yoshida, H.: Grain-128AEAD - a lightweight AEAD streamcipher. NIST Lightweight Cryptography, Round 1 Submission (2019)
10. ISO/IEC 29167-13:2015 information technology — automatic identification and data capture techniques — part 13: Crypto suite Grain-128A security services for air interface communications (2015)
11. Mansouri, S.S., Dubrova, E.: An improved hardware implementation of the Grain-128a stream cipher. In: Kwon, T., Lee, M.K., Kwon, D. (eds.) Information Security and Cryptology – ICISC 2012. pp. 278–292. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
12. Proakis, J.G., Manolakis, D.K.: Digital Signal Processing (4th Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2006)
13. Zambreno, J., Nguyen, D., Choudhary, A.: Exploring area/delay tradeoffs in an AES FPGA implementation. In: Becker, J., Platzner, M., Vernalde, S. (eds.) Field Programmable Logic and Application. pp. 575–585. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)