A Compact Implementation of *Edon80*⋆

Markus Kasper, Sandeep Kumar, Kerstin Lemke-Rust, and Christof Paar

Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany
markus.kasper@rub.de, {kumar,lemke,cpaar}@crypto.rub.de

Abstract. *Edon80* is a new stream cipher that is proposed for hardware based implementations (Profile II) in the eSTREAM project. In this contribution, we present a compact implementation of *Edon80* in hardware. This implementation is built by using only one e-transformer for the *Edon80* core. In FPGA, the design can be realized in 52 slices of the smallest Xilinx FPGA Spartan2 device. The ASIC implementation requires an area of 2922 equivalent gates and yields a throughput of 2.18 Mbit/s at 175 MHz. The hardware resources required for the implementation of *Edon80* can be further reduced to less than 1850 equivalent gates if a micro-controller is available to input the initialization sequence.

Keywords: Edon80, eSTREAM, stream ciphers, hardware implementation.

1 Introduction

Stream ciphers have traditionally a wide use and there is still a need for extremely efficient stream ciphers in various applications. Though efficient and trusted block ciphers have been designed by the cryptographic community, the same has not been successful with stream ciphers. It is worth mentioning that the recent European project 'New European Schemes for Signatures, Integrity, and Encryption (NESSIE)', whose main purpose was to recommend cryptographic schemes to provide different security services and be included in standards and products, has decided to recommend none of the submitted stream ciphers because none of them met the rather stringent security requirements put forward by the project. To cater to this need the Information Societies Technology (IST) Programme of the European Commission ECRYPT, the European Network of Excellence in Cryptology set up the eSTREAM project. The project started with a call for stream cipher primitives in November 2004 [1]. Stream ciphers were solicited which satisfied any one of the two properties known as PROFILES:

– PROFILE 1 : Stream ciphers for software applications with high throughput requirements.

⋆ The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT, the European Network of Excellence in Cryptology.

– PROFILE 2 : Stream ciphers for hardware applications with restricted resources such as limited storage, gate count, or power consumption.

The call resulted in 34 candidates from academia and industry. After an initial Phase 1 of the project, 28 ciphers have been short listed for Phase 2, where the security and performance analysis is being done.

Edon80 [2] is one of the stream ciphers that has been proposed for hardware based implementations (Profile 2). For evaluating the performance of Profile II candidates compactness (area) and performance (throughput) of an implementation are the most important categories [4].

Whereas for some other stream cipher candidates for Profile II, first implementation results were presented at SASC 2006 [5, 6], independent implementation results of *Edon80* are not provided yet. In [3], the designers of *Edon80* give some figures for an implementation, i.e., an estimation of 7,500 gates.

In this contribution, we summarize the possible implementation options towards compactness for *Edon80* in hardware and realized a compact implementation.

The paper is organized as follows: in Section 2 and 3 we give a brief description of the hardware design of *Edon80* as it was proposed by its inventors in [2]. In Section 4 we study different implementation choices aiming at minimizing area requirementions of *Edon80* . Section 5 describes in detail the design of our compact implementation of *Edon80* and Section 6 presents the results achieved for both Xilinx FPGAs and an ASIC implementation. Finally, before concluding in Section 8, Section 7 indicates further improvements, i.e., one may include several e-transformers in our design to enhance the throughput of the implementation.

2 Description of *Edon80*

Edon80 is a binary additive stream cipher. Fig. 1 shows the global schematic representation of the *Edon80* architecture as proposed by the authors. *Edon80* generates the bit keystream z_i based on the *secret key* k and the *initialization vector* iv as inputs. The keystream is XORed to the *plaintext* input m_i during encryption (or the *ciphertext* c_i during decryption) to generate the output stream.

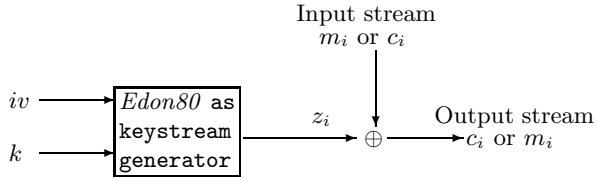

Fig. 1. Graphical representation of *Edon80* as binary additive stream cipher

The behavioral description of *Edon80* is shown in Fig. 2.

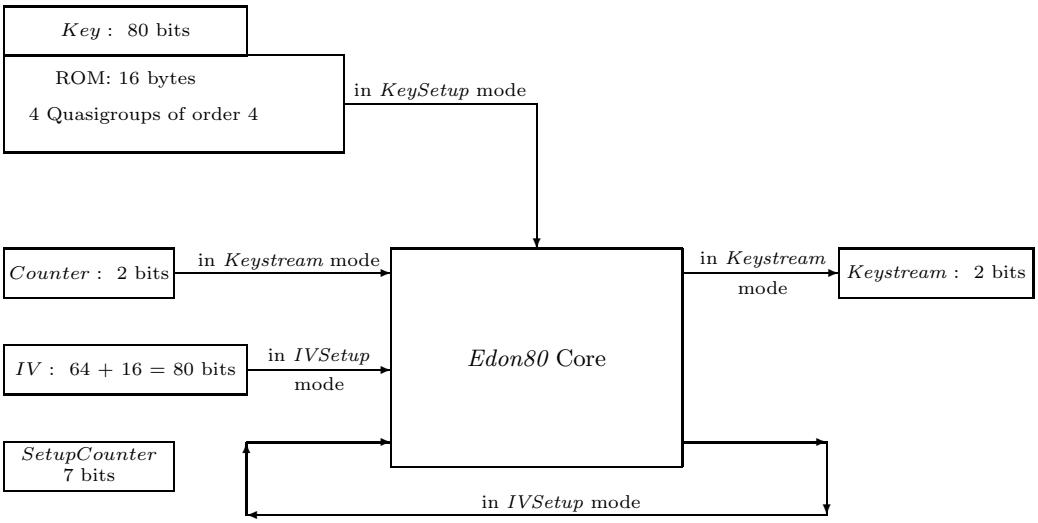

Fig. 2. *Edon80* components and their relations

Besides the *Edon80* Core (which is described later) the cipher requires the following additional resources:

1. A register Key of 80 bits to store the actual secret key.
2. A register IV of 80 bits to store a padded initialization vector.
3. An internal 2-bit *Counter* as a feeder for the *Edon80* Core in *Keystream* mode.
4. A 7 bit *SetupCounter* that is used in *IVSetup* mode.
5. A $4 \times 4 = 16$ bytes ROM bank where 4 quasigroups of order 4 are stored, indexed from $(Q, \bullet_0)$ to $(Q, \bullet_3)$.

The internal structure of *Edon80* Core, as shown in the Fig. 3, is a pipelined architecture of 80 simple 2-bit transformers called *e-transformers*. The 80 *e-transformers* indexed from 0 to 79, are cascaded in a pipeline, one feeding the other.

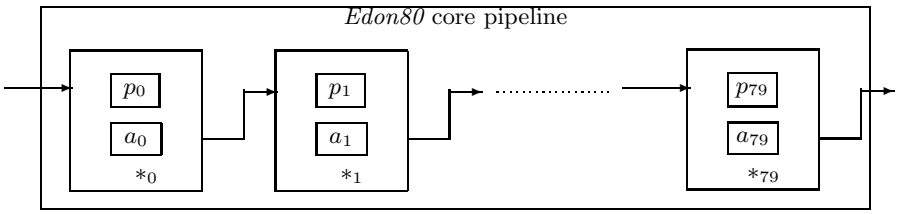

Fig. 3. *Edon80* core of 80 pipelined e-transformers.

The schematic view of an e-transformer is shown in Fig. 4.

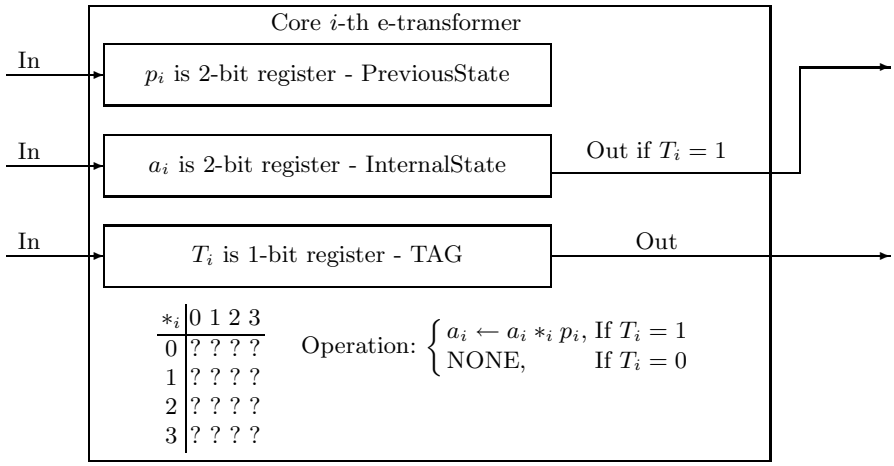

Fig. 4. Schematic representation of a single e-transformer of *Edon80*.

The structure that performs the operation $*_i$ in the e-transformer is a quasigroup operation of order 4. We will henceforth refer to an e-transformer by its quasigroup operation $*_i$. The two 2-bit registers inside every e-transformer (p_i and a_i) are used as the operands by which the new value of a_i is determined according to the defined quasigroup operation $*_i$ for that particular e-transformer. For different e-transformers, there are different possible quasigroup operations defined, chosen from a set of 4 predefined quasigroups of order 4. Every e-transformer is designed to have one tag-bit T_i, which controls if the e-transformer computes the next value a_i or does nothing. All the 80 e-transformers work in parallel to calculate their new value of a_i (if the tag permits that) and then passes the new values of a_i to the next neighboring register p_{i+1} in the pipeline. If the tag forbids the calculation of a_i, the only value that is transferred to the neighboring element is the value of the tag T_i.

3 Modes of Operation

Edon80 works in three possible modes:

1. *KeySetup* mode
2. *IVSetup* mode, and
3. *Keystream* mode.

We give a description of the operation in each of these modes in following subsections.

3.1 *KeySetup* mode

When working in *KeySetup* mode, after transferring the 80 bits to the register Key, the key is virtually divided into forty 2-bit consecutive values. This implies that we represent Key as $K_0K_1 \cdots K_{39}$, where each K_i is 2 bits long, and hence a value ranging between 0 and 3. Based on these values of K_i, in the *KeySetup* mode, every working quasigroup operation $(*_i,\ i = 0, 1, 2, \ldots, 79)$ is assigned by the following rule:

$$(Q, *_i) \leftarrow \begin{cases} (Q, \bullet_{K_i}) & 0 \le i \le 39 \\ (Q, \bullet_{K_{i-40}}) & 40 \le i \le 79 \end{cases}$$

3.2 *IVSetup* mode

In the *IVSetup* mode, after transferring 64 bits to the register IV, the IV is padded with a 16-bit constant $(1110010000011011)_2$, whose interpretation as a concatenation of 2-bit variables gives the string: $(32100123)_4$. We represent the padded IV as a concatenation of forty 2-bit variables v_i, i.e., $IV = (v_0v_1 \cdots v_{31}\ 3\ 2\ 1\ 0\ 0\ 1\ 2\ 3)_4$, where $v_0v_1 \cdots v_{31}$ are the originally transferred 2-bit variables. *IVSetup* is then performed as follows:

- *Initialization:* Make the following assignments:

$$\begin{array}{ll} T_i \leftarrow 0 & i = 0, ..., 79 \\ a_{39-i} \leftarrow v_i & i = 0, ..., 39 \\ a_{79-i} \leftarrow K_i & i = 0, ..., 39 \end{array}$$

- *Cycle 0:* Set the tag T_0 to 1 and feed the register p_0 with the value of K_0. Recall that the value of a_0 is assigned to v_0 by the initialization, so the new value of a_0 will be $v_0 *_0 K_0$.
- *Cycle 1:* The new values of a_0 and T_0 are sent to the second e-transformer $*_1$ and the register p_0 is loaded with K_1. In such a way the assignments $p_0 \leftarrow K_1,\ p_1 \leftarrow a_0,\ T_1 \leftarrow T_0$ are made.
- *Cycles 2 - 79:* In the next 78 clock cycles, the register p_0 of the *Edon80* Core is feeded with the values in the order: $K_2,\ K_3,\ \ldots,\ K_{39}$ and followed by $v_0,\ v_1,\ \ldots,\ v_{39}$. Hence, each of the e-transformers $*_2,\ \ldots,\ *_{79}$ begin to start working consecutively .
- *Cycle 80:* The tag T_0 is set back to 0, and the content of the register a_{79} is fed into the register a_0. Notice, that after this cycle the e-transformer $*_0$ will stop and the value of the register a_0 will be preserved. (That value is in fact the initial internal state of the register a_0 for the *Keystream* mode.)
- *Cycle 81 - 159:* In the next 79 cycles, all of the e-transformers $*_1, \ldots, *_{79}$ stop one after another. When the register $*_i$ stops, the content of the register a_{79} is fed into the register a_i. (Notice that the values of the registers $a_0, a_1, \ldots, a_{79}$ are the initial internal states for the *Keystream* mode.)

For the concrete realization of all cycles in the *IVSetup* mode, the internal 7 bit register *SetupCounter* and a related logic controlling its values (reset to 0 upon reaching the value of 79) are needed.

3.3 *Keystream* mode

To start the *Keystream* mode, the *Counter* is reset to 0 and the value of T_0 is set to 1. In the *Keystream* mode, the *Edon80* Core is fed by the register *Counter*, which increases its value every cycle. After a latency of 80 cycles, a bit keystream starts to flow out of the last e-transformer, i.e., from the 2-bit register a_{79}. It has to be stressed that the bit keystream generated consists of only every second value that comes out of a_{79}, to maintain the security of the cipher.

4 Hardware Minimization for *Edon80* implementation

As the aim of our implementation of *Edon80* is to minimize the area, the first step we did was to identify possibilities to reduce the gate count of the architecture. In this section, we describe the different steps we performed to minimize the required hardware effort to implement the cipher.

4.1 The Quasigroup

First, we notice that the quasigroup operation is a structure that is replicated 80 times. Therefore, to achieve a small implementation, it is necessary to reduce the gate count of these quasigroup operations to a minimum. There are many possible ways to implement a quasigroup. To find the best solution, we implemented and evaluated different quasigroup architectures as mentioned below:

- The quasigroups were designed as a ROM using a corresponding decoder structure.
- A RAM approach was used, where every stage had knowledge about the quasigroup it would use.
- The RAM architecture was extended such that two e-transformers were used accessing the same RAM through different ports.
- A hierarchy of multiplexers for K_i, p_i and a_i was built to select the corresponding output bits of a quasigroup operation.
- An approach using logic gates to decide the outcome of the quasigroup was implemented.
- Finally, we implemented a bitslice like logic structure, that uses binary logic but exploits the combinations of input bits that occur more than once.

Among these different possibilities, the last approach lead to the smallest design we could find for a quasigroup. Implementing a quasigroup as a logic design does not require the additional $4 \times 4 = 16$ bytes of ROM bank as proposed.

4.2 The e-transformer

This structure is also replicated 80 times and therefore requires a compact implementation to avoid unnecessary waste of resources. First, we note that the p_i value of an e-transformer is the same as the a_{i-1} value already stored in the

preceding e-transformer. Therefore, we can substitute p_i by a_{i-1} and remove all registers required to store the p_i values. A disadvantage about the e-transformer is the need of a 'writeback' mechanism during *IVSetup* which costs a lot of gates. Another issue we noticed is, the tag register required in each stage. Later in this paper, we will show how to completely eliminate both of them from the hardware design of *Edon80*.

4.3 IV register, Key register and the 2-bit counter

Both of these registers are unnecessary if we write the IV and Key directly to the corresponding internal flipflops (FFs), as required by the *Keysetup* and *IVSetup*. The 2-bit counter is also unnecessary if we consider the fact that we already have an idle *SetupCounter* in the proposed design that can be reused.

4.4 The pipeline concept

To minimize the area of an *Edon80* implementation, one can completely eliminate the pipeline concept of *Edon80*. But this decision dramatically reduces the throughput of an implementation, because one now needs 80 clock cycles per key bit output, whereas one would reach 1 bit per cycle throughput with the fully pipelined implementation. Thus, instead of the 80 separate e-transformers, we decided to use just one single e-transformer. This idea of a sequential implementation is explained further in Section 5. Using such a non-pipelined concept, the size of the quasigroup operation is negligible compared to the area of the cipher implementation.

4.5 The *IVSetup* procedure

The *IVSetup* suggested in Section 3 is complicated and needs a lot of hardware resources. It is, however, possible to perform the same initialization in a simpler fashion (as suggested by the software implementation by the inventors of *Edon80*) using the following sequence:

- Initialize a_0 to a_{39} with K_0 to K_{39}, and a_{40} to a_{79} with v_0 to v_{39}.
- Set p_0 input to v_{39} and use K_0 for all stages.
- Subsequently update a_0 to a_{79} using key K_0 and the value of the previous stage as p_i.
- After 80 clock cycles, a_{79} is updated and the same sequence is performed again with the p_0 input set to v_{38}, and the key set to K_1.
- This process is repeated using the same pattern: update p_0 and the key to be used; then update all states during the next 80 cycles; when finished, select the next set of p_0 and key. The sequence used for the p_0 input is: v_{39}, v_{38}, ..., v_0, K_{39}, K_{38}, ...,K_0. The sequence used for the key is: K_0, K_1, ..., K_{39}, K_0, K_1, ..., K_{39}.
- When all states are updated the 80th time, the *IVSetup* is completed.

This way, the initialization of the states requires neither tag registers, nor a 'writeback' mechanism.

5 Implementation Details of *Edon80*

The design of *Edon80* was developed in VHDL such that is could be synthesized both for FPGA and ASIC. First, we present the hierarchy of the non-pipelined sequential implementation of *Edon80* as described in Section 4 and then describe each of the components. The throughput of this implementation is 1/80 bit per cycle as the keystream outputs every second 2-bit outcome and it takes 2x80 cycles to generate a pair of keybits.

The implementation hierarchy is as follows:

- a block *Edon80*, which serves as an interface to external circuits and connects all the internal components of the *Edon80* algorithm to each other,
- a block *quasigroup*, which performs the quasigroup operations based on the inputs: key K, a and p,
- a block *etransformer*, which represents an e-transformer and serves as a wrapper for a child quasigroup operation,
- a 2x80bit shift register *aSHR*, which contains the states: a_0 to a_{79},
- a 2x40bit shift register *kSHR*, which contains the keys: K_0 to K_{39},
- a 2x80bit shift register *initSHR*, which contains the necessary inputs to perform the initialization sequence required by the specification of *Edon80* [2],
- a block *control*, which manages the counters and provides all internal control signals,
- a block *outputprocessor*, which filters keybits from other output.

5.1 The Quasigroup Block

This block with inputs K, a and p, and output a_out implements the lookup process for a quasigroup operation. We took each quasigroup table and mapped it to its logic, trying to exploit redundant signal combinations to reduce the gate count. Afterwards, a multiplexer is utilized to select the right output with respect to key K. We made these optimizations by hand, and therefore one could improve the gate count using an optimization programm to find a better logic structure than ours. However, we believe to be reasonably close to the best possible implementation, and with just one quasigroup in our design, we decided not to spend any more effort into further optimization of this block. The logic functions we used to determine the right output are:

$$
\begin{aligned}
f_0 &\leftarrow a_0 \oplus p_0 \\
f_1 &\leftarrow a_1 \oplus p_1 \\
f_2 &\leftarrow \overline{a_0} \\
f_3 &\leftarrow f_1 \cdot f_2 \\
f_4 &\leftarrow a_1 \oplus p_0 \\
f_5 &\leftarrow \overline{a_1} \\
f_6 &\leftarrow f_0 \oplus p_1 \\
f_7 &\leftarrow \overline{f_6} \\
f_8 &\leftarrow \overline{a_0 \oplus p_1} \\
h_{K0} &\leftarrow f_3 + a_0 \cdot (f_1 \oplus p_0) \\
l_{K0} &\leftarrow f_0 \\
h_{K1} &\leftarrow f_4 \cdot a_0 + \overline{f_1 \cdot f_2} \\
l_{K1} &\leftarrow f_2 \cdot f_4 + a_0 \cdot f_1 \\
h_{K2} &\leftarrow a_1 \cdot f_7 + f_0 \cdot f_5 \\
l_{K2} &\leftarrow f_5 \cdot f_7 + \overline{a_1 \cdot f_0} \\
h_{K3} &\leftarrow \overline{f_4} \\
l_{K3} &\leftarrow a_1 \cdot f_6 + f_5 \cdot f_8
\end{aligned}
$$

Here, $\cdot$ denotes logic AND, $\oplus$ denotes logic XOR, $+$ denotes logic OR, and a bar $\overline{X}$ denotes negation. Furthermore, the indices of p and a denote the higher bit (if index is '1') and lower bit (if index is '0'). A multiplexer then selects the bit pairs for the new a. Hence,

$$
a\_out = \begin{cases} h_{K0}\ l_{K0} & \text{when } K = 0 \\ h_{K1}\ l_{K1} & \text{when } K = 1 \\ h_{K2}\ l_{K2} & \text{when } K = 2 \\ h_{K3}\ l_{K3} & \text{when } K = 3 \end{cases}
$$

For example if K is $(00)_2$ the two output bits h_{K0} and l_{K0} are selected for a_out.

5.2 The Etransformer Block

The block *etransformer* includes the quasigroup block. This entity has many input ports:

– *p_in*: This port provides the *etransformer* with the 'external' p input. We use this port in our implementation to feed a_0 with the *IVSetup* sequence every 80th cycle.

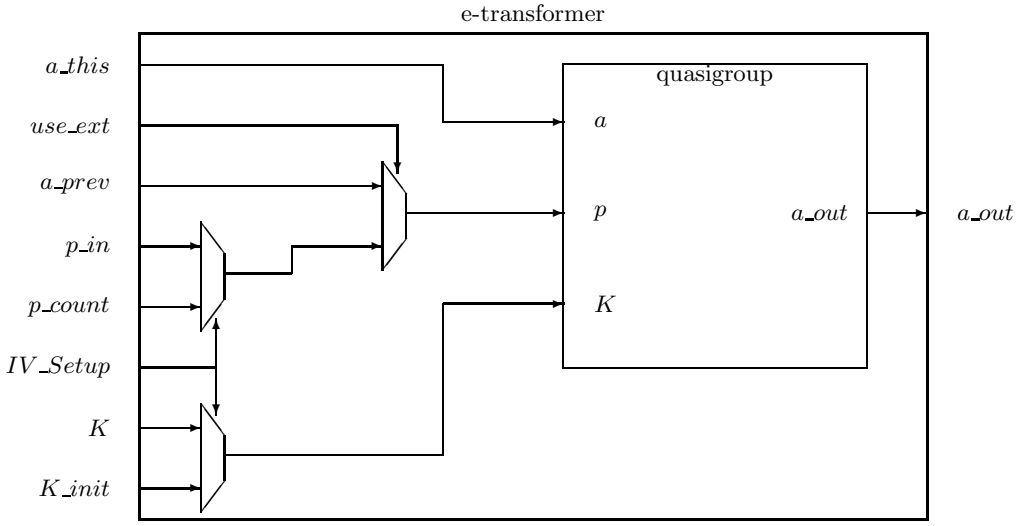

Fig. 5. e-transformer design

- *p_count*: The 2-bit counter, which serves as an external input to the first pipeline stage during *Keystream* mode, is fed to this port.
- *a_prev* and *a_this*: These ports input the p and a values during *Keystream* mode, if the e-transformer does not update a_0.
- K and *K_init*: While K holds the keybits corresponding to the stage the e-transformer is working in, *K_init* provides the key to be used during *IVSetup*.
- *IV_Setup*: This signal generated by the control unit tells the e-transformer if the algorithm is working in *IVSetup* mode.
- *use_ext*: If this signal goes to high, the e-transformer is working on *a_0*, and should therefore use an external p as an input to the quasigroup. This signal ensures a_0 is always using the external value from one of the p ports.
- The output *a_out* of this entity is the corresponding lookup value of the quasigroup that will be used to update the a state the e-transformer is working on.

Internal multiplexers select the right inputs to the quasigroup. If the *use_ext* signal is low, then p of the quasigroup is fed with the *a_prev* signal, which is the a used in the previous stage. If *use_ext* is high this implies that the e-transformer is working on a_0 and should therefore use an external input for p. During *IVSetup* (determined by *IV_Setup* high), the e-transformer selects the *p_in* port as an external input to p_0. Otherwise, the 2-bit counter at *p_count* will be used as p_0. Another multiplexer ensures that quasigroup K uses *K_init* while *IV_Setup* is high, and the regular K port of the e-transformer when *IV_Setup* is low. The a value of the quasigroup is *a_this* in all cases and the output *a_out* is forwarded to the e-transformer output port right away.

Note that this e-transformer contains only logic and does not store any values in it.

5.3 The shift registers

This hardware design utilizes three shift registers (SHR). The key shift register *kSHR* holds the 40 keybit pairs, and once initialized, rotates the keys every cycle and provides the content of the K_0 register to the e-transformer. The a shift register *aSHR* holds the 80 a-states of the algorithm and performs nearly the same operation, except instead of rotating, it uses the output of the quasigroup as the new input. The third shift register is the *initSHR*. This shift register holds an 80 symbol sequence of bit pairs of IV and K, which is needed to initialize the internal *Edon80* states. While the other two shift registers change every cycle, the $initSHR$ updates only every 80th cycle. Details of each SHR entity are given below.

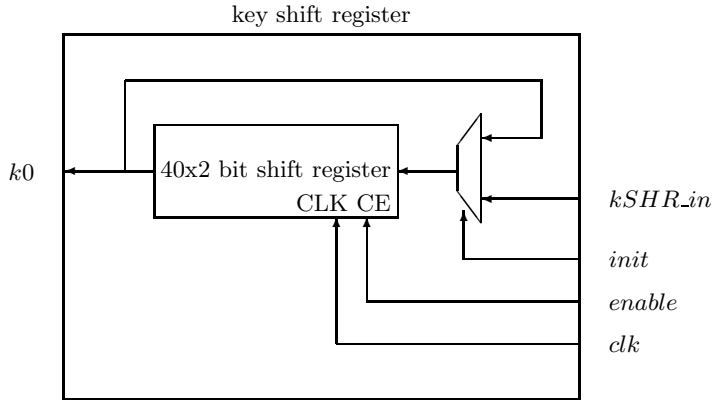

Fig. 6. key shift register design

Key shift register - kSHR: This entity has the 4 input ports: *enable*, *init*, $kSHR\_in$ and *clk*, and one output $k0$. While the *enable* (high) signal determines whether the shift register will respond to clock events with a shifting operation or if it keeps its current state, the *init* signal is responsible to select between *Initialisation* mode (high) and normal mode (low). During *Initialisation* mode, the *kSHR* acts as a simple shift register with $kSHR\_in$ as input and the discarded output $k0$. In normal mode, the shift register rotates its contents using the $k0$ as an input. This structure allows initializing the kSHR with a given sequence and let it rotate afterwards.

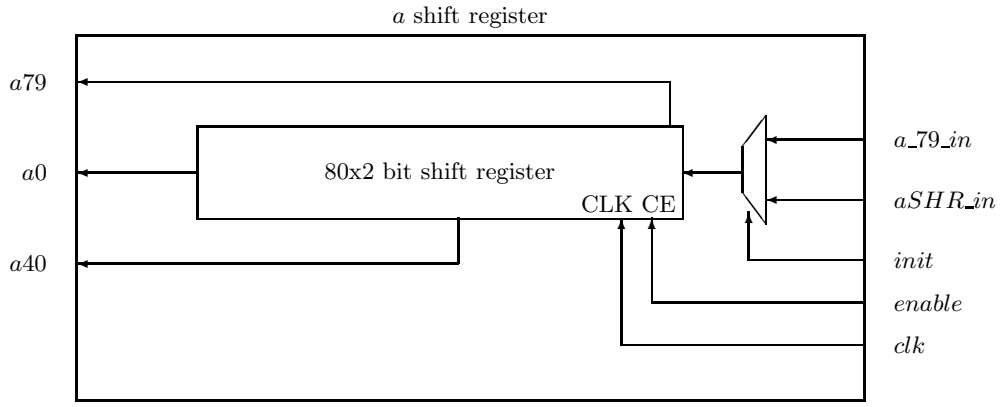

Fig. 7. *a* shift register design

***a* shift register - aSHR:** The *aSHR* is similar to the *kSHR*, with the input ports: *enable*, *init*, $aSHR\_in$, a_79_in and *clk*, and the output ports: $a0$, $a40$ and $a79$. The *enable*, *init*, $aSHR\_in$ and *clk* inputs function similar to the corresponding inputs in the *kSHR*, except that instead of rotating when *init* is low, the *aSHR* uses a_79_in as the input to the shift register. In fact, the signal arriving at this port is the a_out signal from the e-transformer which is used to update the state a_0 that is shifted to a_{79}. The output $a0$ is used as a_this for the e-transformer, and the $a79$ provides the a_prev.
The idea here is to connect two consecutive states to the e-transformer and update the value of the one having the role of a_i. At the same time, we shift the shift register so that we have to write the output to a_{79} instead of to a_0. This way, we move subsequently along the states updating one after another. The e-transformer will take care of providing the real a_0 with the correct p_0 when updating it. The $a40$ port is an additional output that is connected to $kSHR\_in$. This way we can initialize a_0 to a_{39} and K_0 to K_{39} at the same time, as they get the same values anyway.

Init shift register - initSHR: The main structure of this shift register is identical to the one just introduced. The input ports *enable*, *init*, $initSHR\_in$ and *clk*, and the output ports $init0$ and $init40$ provide the same functionality as the corresponding ports of the other shift registers. Here, *init* (low) switches to use $init40$ as input to the shift register. The output $init0$ is connected to p_in of the e-transformer, and the second output $init40$ is used as K_init. The port $init0$ also serves as the input to $aSHR\_in$, and $initSHR\_in$ itself is connected to the external port *Input* of the *Edon80*. Using $init40$ to repeat the last 40 values provides the correct K_init keys during *IVSetup*, where we need a sequence of K_0 to K_{39} and another K_0 to K_{39}. At the same time the correct K values arrive as p_in at the end of *IVSetup* without any additional hardware effort.

Note that using this setup during initialization, all SHRs are connected to one long shift register leading to a very simple and small structure.

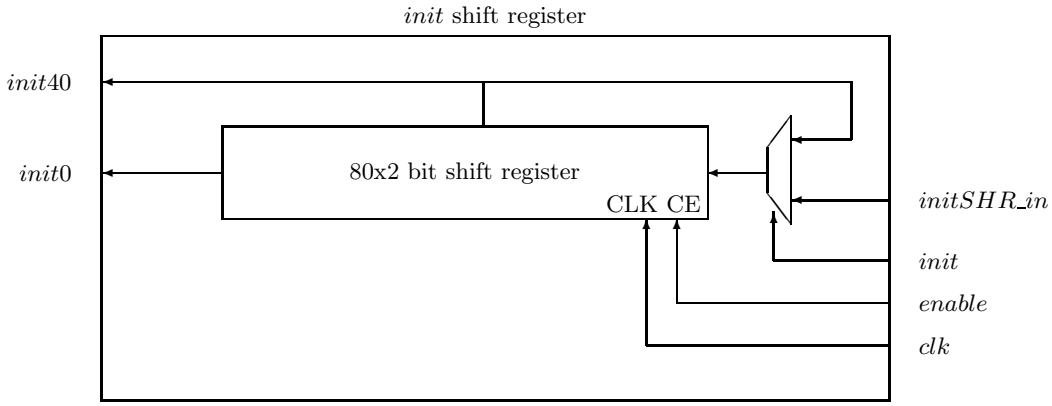

Fig. 8. *init* shift register design

5.4 The Outputprocessor Block

This entity uses the *writeout* signal provided by the control unit to filter the key-bits from other bit-pairs. A 2-bit register stores the latest two key-bits and delivers them to the external interface as *Output*. This *Output* is updated with the next bit-pair when *writeout* is high.

5.5 The Control Unit

The control unit uses two 7-bit counters: *counter* and *counter*2 enable this entity to generate all required control signals for the algorithm. The control unit is provided with the inputs: *init*, *T_in* and *clk*, and generates the output flags: *enable*, *IV_Setup*, *use_ext*, *ready* and *writeout*, and outputs another 2-bit value *p_count*. When *init* is high, the *Edon80* is working in initialization mode. When *init* switches to low, the *Edon80* will start the *IVSetup* and set the *IV_Setup* signal to high. During *IVSetup* and *Keystream* mode, the e-transformer needs information about which state is the first one. Therefore the control unit sets *use_ext* to high every 80th clock cycle. Once the *IVSetup* has finished, the corresponding signal goes back to low and the e-transformer now works in *Keystream* mode. Here, we need the 2-bit input counter for the pipeline *p_count* and another signal called *writeout* to determine whether the current e-transformer output belongs to the keystream or not. An additional output flag *ready* sends a logic 1 to the external circuit every time *Edon80* has written the next two keystream bits to its *Output* port. The external input port *T_in* of the *Edon80* is used to pause

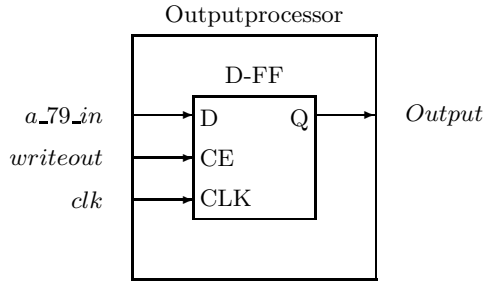

Fig. 9. Outputprocessor design

the generation of keystream bits when set to low. The internal signal *enable* is set to the value of T_in only in the *Keystream* mode.

All of these signals are generated by the two counters, that are nested into each other. As long as *enable* is high, *counter* is increased by one every clock cycle, counting from 0 to 79. Then it is reset to zero again and will continue recounting. Every time *counter* is reset, the *counter*2 is increased by one and also counts from 0 to 79. Here the reset also triggers the *initdone_int* signal to high to indicate that *IVSetup* has finished. The following statements show how the described signals are generated.

- *use_ext* is high whenever *counter* is 0.
- *IV_Setup* is high when neither *init* is high nor *initdone_int* is high.
- *enable* is high as long as *initdone_int* is low. When *initdone_int* is high, *enable* follows T_in.
- *p_count* reuses the lower 2 bits of *counter*2. This works because 80 is an integer multiple of 4 and because Initialisation and *IVSetup* also need an integer multiple of 4 clock cycles.
- *writeout* goes to high when *counter*1 is 79 (the e-transformers output belongs to state a_{79}) and *initdone_int* is high (indicating we are in keystream mode) and the lowest bit of *counter*2 is high (to select just every second outcome as keybits).
- *ready* is generated by a single D-FlipFlop, which follows *writeout* with a one clock cycle delay.

5.6 The *Edon80* container

The *Edon80* entity connects all these blocks together to build the *Edon80* algorithm. The following description shows how this architecture works.

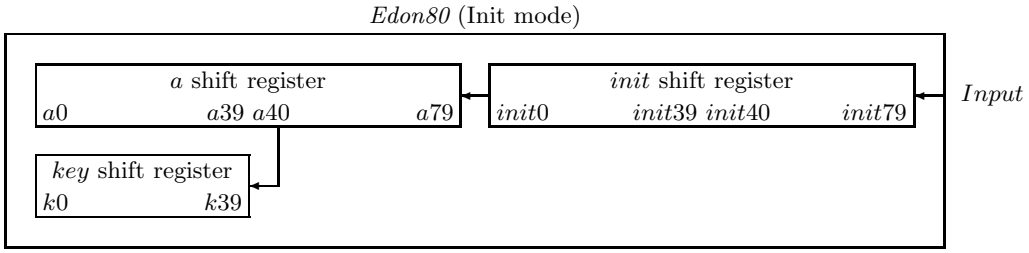

Fig. 10. *Edon80* during Initialization

Initialization: We start the *Edon80* algorithm by performing an initialization to all shift registers and counters. This requires the following interaction:

- set the *init* signal to logic '1' and send K_0 to port *input*.
- during the following clock cycles send K_1 to K_{39} and afterwards v_0 to v_{39} to the input port. Make sure to use the correct padding sequence of 32100123 for the last v values.
- during the next 80 clock cycles send the same values in a backward order to the input port, starting with v_{39} and ending with K_0.
- set the *init* signal back to logic '0'.

During Initialization the shift registers are connected to each other as shown in Figure 10. After this initialization phase, the states of the registers are as follows:

- Both a_0 to a_{39} and K_0 to K_{39} hold the values K_0 to K_{39}.
- a_{40} to a_{79} hold v_0 to v_{39}.
- The Initregister holds v_{39} to K_0.
- All counters are reset to 0 and the flag *initdone_int* is reset to logic '0'.

When *init* goes to zero the *Edon80* starts the *IVSetup*.

IVSetup: During *IVSetup* mode the e-transformer uses *initreg*40 as a Key and *initreg*0 as external *p_in* input. $aSHR0$ and $aSHR79$ are used as a and p of the quasigroup. The next cycle, when the shift register moves all states by one symbol, the result of the operation is written to state $aSHR79$. The e-transformer starts with the p input from the Initregister0 and the content of $aSHR0$ to calculate the first value. In the next clock cycle, the shift register

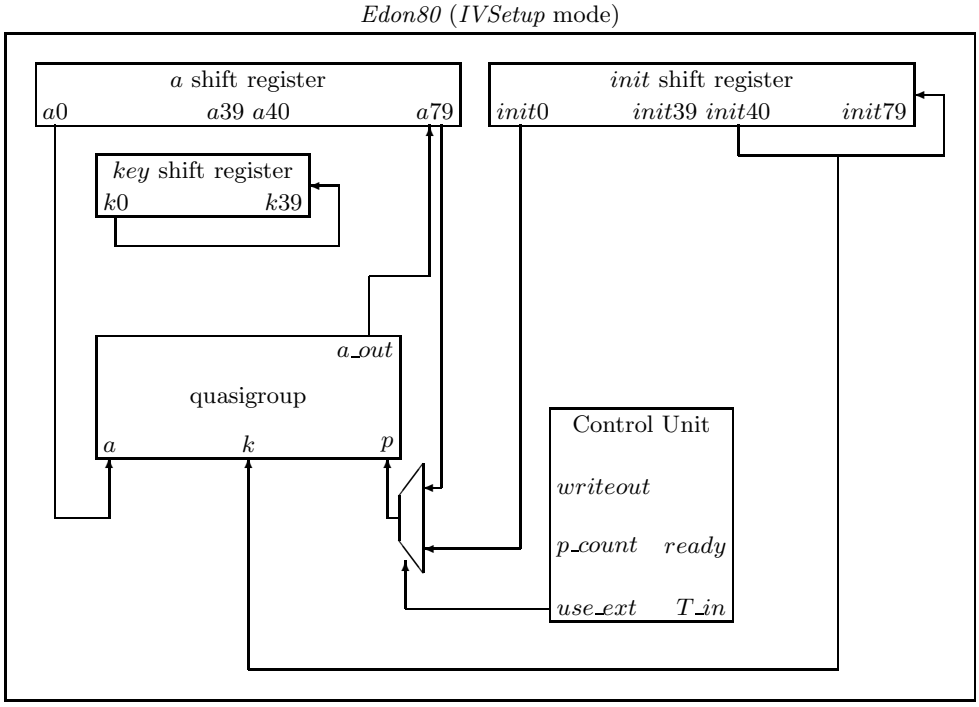

Fig. 11. *Edon80* during *IVSetup*

moves one symbol and $aSHR79$ gets the result of the e-transformer as the new content. Now $aSHR79$ holds the updated state a_0 of the *Edon80* algorithm. The ports of the e-transformer are now connected to the updated value of a_0 in $aSHR79$ and a_1 in $aSHR0$. The quasigroup lookup is stored in $aSHR79$ again when the next clock cycle arrives. This way, we subsequently update all 80 a values until a_0 arrives at the e-transformer again. The control unit tells the initregister to cycle one symbol and the e-transformer to use the input from the initregister instead of $aSHR79$ by setting *use_ext* to high again. With the next key and new external *p_in*, all states are updated again. This is repeated for the next 80 rounds, or in other words: until every symbol in the Initreg has been the p input once. At this point the control unit indicates that *IVSetup* has finished and switches to *Keystream* mode.

Keystream mode: In *Keystream* mode, the input from the initregister is replaced by the 2-bit counter *p_count* generated by the control unit. The key is now connected to the key shift register and rotates with every clock cycle. The *use_ext* signal by the controller is used again to determine when the e-transformer has to use the external *p_in* instead of the $aSHR79$ value. In addition, the output filter now gets active and sorts out which e-transformer outputs are to be forwarded

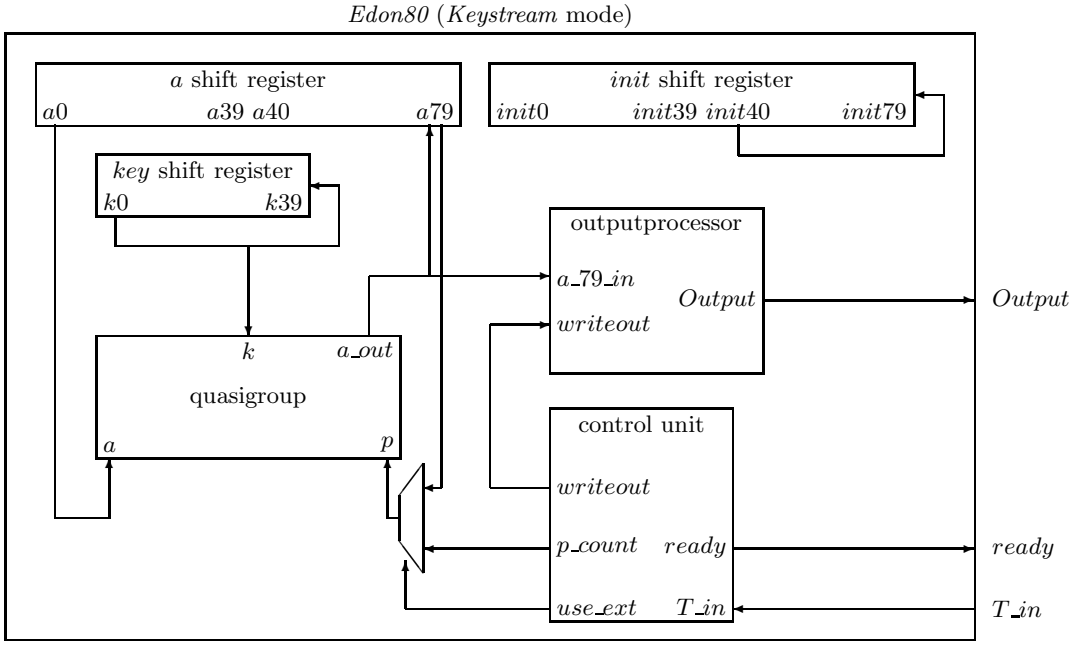

Fig. 12. *Edon80* during *Keystream*

to the external circuit. Each time it updates, the *ready* signal of *Edon80* is set to high for one clock cycle. In addition, the input signal T_in, which allows to pause the algorithm from generating more keystream bits, gets into effect now. When T_in is set to '0', the algorithm will either stop right away, or if it is in *IVSetup* or *Initialization*, it will continue until *IVSetup* has finished and holds afterwards. Whenever T_in is set to logic '1' again, it will continue generating the keystream (so '1' should be the default input to this port). To start over again, one just needs to set init to logic '1' again and enter a new initialization sequence.

6 Results

The design written in VHDL was synthesized using Xilinx ISE 8.1i tools, and simulated and verified with Mentor Graphics' Modelsim XE III. We mapped the implementations to Xilinx Spartan and Virtex FPGAs and compared the results with the estimated computational efficiency in hardware [3], proposed by the inventors of *Edon80*. Table 1 shows the results we achieved.

Table 1. Implementation Results for Xilinx FPGAs.

FPGA Type	MHz	Throughput	Logic Utilization (Utilized/Total)
Spartan2 Target Device: xc2s15 Target Speed: -6	106	1.33Mbit/s	Slices: 52/192 Slice Flip Flops: 27/384 4 input LUTs*: (49+30)/384
Spartan3 Target Device: xc3s50 Target Speed: -5	149	1.87Mbit/s	Slices: 50/768 Slice Flip Flops: 27/1536 4 input LUTs*: (49+30)/1536
Virtex2 Target Device: xc2v40 Target Speed: -5	209	2.62Mbit/s	Slices: 49/256 Slice Flip Flops: 27/512 4 input LUTs*: (52+30)/512
Virtex4 Target Device: xc4vlx15 Target Speed: -12	286	3.58Mbit/s	Slices: 45/6144 Slice Flip Flops: 27/12288 4 input LUTs*: (50+30)/12288

* - (LUTs used as logic + LUTs used as Shift registers)/Total

With 2 bits of output every 160 cycles, we achieve a throughput of 1 bit per 80 cycles.

The same design was synthesized for ASIC using the Synopsys Design Compiler (V-2004.06-SP1) tools for AMI Semiconductor $0.35\mu m$ CMOS technology. Mapping this design to an ASIC, the design required an area of 2922 equivalent gates at 175 MHz (equivalent output rate of 2.18 Mbit/s). The area requirements of the individual blocks of the *Edon80* are shown in Table 2.

Table 2. Equivalent gate count for the main blocks of *Edon80* in ASIC.

Block	Reference Name	Area (equivalent gates)
Key shift register	kSHR	537.5
a shift register	aSHR	1070.3
Init shift register	initSHR	1072.0
e-transformer	etransformer	59.7
Control Unit	control	164.7
Outputprocessor	outputprocessor	13.3

The *Edon80* turned out to be remarkably scalable. As we can easily see, we just need additional shift registers and small adjustments to the control unit to

increase key and IV lengths. At the same time, one can freely choose the number of quasigroups that should work on the stages at the same time, to increase the level of parallel processing.

7 Further Improvements

The pipelining concept: Instead of implementing a pipelined *Edon80*, our approach has been to use just one single e-transformer and additional logic to implement the design. This aims to minimize the overall gate count, but also leads to the worst performance. However, parallel processing is still possible with this concept. Several separated e-transformers acting on the same 'states' can be implemented to simulate an 80 stage pipeline. For example, one could use two e-transformers updating the states independently to double the throughput. The case with just one e-transformer is the extreme case of this concept. In case of 10 e-transformers the throughput is then 0.125 bits per cycle which is about the same throughput that was achieved with the smallest AES ASIC design by Feldhofer et. al. [4] and might be a good choice for performance benchmarking.

The enable signal: The *enable* signal could be removed from the design in case it is not required to pause the keystream generator at any time.

The initialization register: One can remove the init register from the design and let a micro-controller take care of performing the right initialization sequence with the correct timing. Our ASIC implementation results in Table 2 showed that the initregister takes more than a third of the overall area needed by *Edon80*. The removal of the init register and the corresponding simplifications of the control unit (i.e. smaller *counter*2) can reduce the hardware design to less than 1850 equivalent gates.

8 Conclusion

This contribution presents a compact implementation of *Edon80* in hardware. In FPGA, the design can be realized in under 52 slices and hence can easily fit the smallest Xilinx FPGA Spartan2 devices. The ASIC implementation requires an area of 2922 equivalent gates which is less than the area of the smallest AES implementation known [7]. The hardware resources required for the ASIC implementation of *Edon80* can be further reduced to less than 1850 equivalent gates if a micro-controller is available to input the initialization sequence. Our implementation shows that *Edon80* is an interesting candidate for compact implementations in eSTREAM Profile II that additionally offers good scalability.

References

1. The eSTREAM Call for Stream Cipher Primitives, http://www.ecrypt.eu.org/stream/call/

2. Danilo Gligoroski, Smile Markovski, Ljupco Kocarev and Marjan Gusev. Algorithm Description of Edon80, available at http://www.ecrypt.eu.org/stream/ciphers/edon80/edon80.zip
3. PartB6: Estimated computational efficiency in software and hardware, part B6 of [2]
4. L. Batina, S. Kumar, J. Lano, K. Lemke, N. Mentens, C. Paar, B. Preneel, K. Sakiyama and I. Verbauwhede. Testing Framework for eStream Profile II Candidates. SASC 2006 Stream Ciphers Revisited, Workshop Record, p. 104–112.
5. Frank K. Gürkaynak, Peter Luehthi, Nico Bernold, René Blattmann, Victoria Goode, Marcel Marghitola, Hubert Kaeslin, Norbert Felber, and Wolfgang Fichtner. Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt. SASC 2006 Stream Ciphers Revisited, Workshop Record, p. 113-124.
6. T. Good, W. Chelton and M. Benaissa. Review of stream cipher candidates from a low resource hardware perspective. SASC 2006 Stream Ciphers Revisited, Workshop Record, p. 125-148
7. Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. AES Implementation on a Grain of Sand. *IEE Proceedings on Information Security*, 152:13-20, October 2005.