

Designing integrated accelerator for stream ciphers with structural similarities

Sourav Sen Gupta · Anupam Chattopadhyay ·
Ayesha Khalid

Received: 15 January 2012 / Accepted: 30 August 2012 / Published online: 28 September 2012
© Springer Science+Business Media, LLC 2012

Abstract To date, the basic idea for implementing stream ciphers has been confined to individual standalone designs. In this paper, we introduce the notion of integrated implementation of multiple stream ciphers within a single architecture, where the goal is to achieve area and throughput efficiency by exploiting the structural similarities of the ciphers at an algorithmic level. We present two case studies to support our idea. First, we propose the merger of SNOW 3G and ZUC stream ciphers, which constitute a part of the 3GPP LTE-Advanced security suite. We propose HiPAcc-LTE, a high performance integrated design that combines the two ciphers in hardware, based on their structural similarities. The integrated architecture reduces the area overhead significantly compared to two distinct cores, and also provides *almost double throughput* in terms of keystream generation, compared with the state-of-the-art implementations of the individual ciphers. As our second case study, we present IntAcc-RCHC, an integrated accelerator for the stream ciphers RC4 and HC-128. We show that the integrated accelerator achieves a slight reduction in area without any loss in throughput compared to our standalone implementations.

This is an extended version of the conference paper [19] by Sen Gupta, Chattopadhyay and Khalid, presented at INDOCRYPT 2011. Summary of changes: Sections 1 and 2 have been considerably revised. Sections 3 and 4 are based on [19], with major revision in Section 4. Sections 4.5, 4.6 and 5 are completely new contributions in this work.

S. Sen Gupta (✉)

Centre of Excellence in Cryptology, Indian Statistical Institute, Kolkata, India
e-mail: sg.sourav@gmail.com

A. Chattopadhyay · A. Khalid

MPSoC Architectures, UMIC, RWTH Aachen University, Aachen, Germany

A. Chattopadhyay

e-mail: anupam@umic.rwth-aachen.de

A. Khalid

e-mail: ayesha.khalid@umic.rwth-aachen.de

We also achieve *at least 1.5 times better throughput* compared to general purpose processors. Long term vision of this hardware integration approach for cryptographic primitives is to build a flexible core supporting multiple designs having similar algorithmic structures.

Keywords Stream ciphers · Integrated accelerator · ASIC · Area efficiency · High throughput · 3GPP LTE-Advanced · SNOW 3G · ZUC · RC4 · HC-128

Mathematics Subject Classification (2010) 94A60

1 Introduction

Stream ciphers hold a major share in the world of symmetric key cryptography, primarily due to their blazing speed of operation and simplicity of design suitable for implementation in both software and hardware. During the last decade, an array of stream ciphers has been developed to cater to the needs of modern day digital communication, both in public and private sectors. Although RC4, designed in 1987, still remains the most widely used stream cipher in the commercial domain, new designs have emerged to address issues of higher security and platform-specific implementation. The main initiative was taken up in the eSTREAM project by European Network of Excellence in Cryptology (ECRYPT), where the goal was to build a portfolio of modern stream ciphers in two categories—software and hardware. The current portfolio of eSTREAM [24] contains HC-128, Rabbit, Salsa20/12 and SOSEMANUK in the software category, and Grain v1, MICKEY v2 and Trivium in the hardware category. Another prime motivation for new stream cipher design has been instigated by the advent of 4G mobile technology. 3GPP LTE-Advanced [2] has been proposed as the leading candidate for 4G mobile broadband services, and it contains the two stream ciphers SNOW 3G and ZUC at the core of its security architecture [1].

It is interesting to note that the design of stream ciphers follow a general trend and frequently abides by either of the following three principles:

- pseudorandom words extracted from a regularly updated pseudorandom state, e.g., RC4, HC-128, Rabbit,
- pseudorandom words extracted from a finite state machine (FSM) that receives its input from a regularly updated linear feedback shift register (LFSR), e.g., SNOW 3G, ZUC, SOSEMANUK, and
- pseudorandom bits as output from a boolean function with inputs drawn from a mutually interdependent or independent combination of LFSR and/or NFSR, e.g., Grain v1, MICKEY v2, Trivium.

Among the stream ciphers mentioned earlier, Salsa20/12 is the only one that follows a block cipher like state update which is completely different from any other design. Apart from that, it seems that modern stream ciphers share a lot of structural similarities, which may potentially be exploited during implementation.

Motivation We consider of the following problem in this direction.

If one wants to implement two or more stream ciphers on the same platform, and if the ciphers share certain structural similarities, then how should one approach the design of an integrated accelerator?

A general solution towards this direction is to incorporate custom instructions for the individual ciphers into a general purpose processor and thus facilitate it to run any cipher independently. However, this kind of an implementation may not always be the best choice in terms of throughput and area, as a general purpose processor with custom instructions do not provide the implementor with full freedom to explore the design space for an optimal solution.

We approach this problem from a completely different angle. Implementing custom instructions in a processor attempts to merge the ciphers from a hardware implementation point of view. We take a step back, and try to merge the ciphers from an algorithmic point of view. Once this is accomplished, one may design an integrated accelerator for the ciphers such that each of the algorithms can be accessed individually. This approach offers the flexibility of

- (a) sharing of resources, both storage and logic,
- (b) throughput vs. area optimization at the base level,
- (c) optimization of mutual critical path, and
- (d) combined protection against fault attacks.

The process of integration at both algorithm and hardware levels produce the best solutions in terms of throughput and area, and provides the designer with handles on both. The hybrid approach tries to find the common computational kernel and add necessary flexibility layer on top of it to cover multiple algorithms. This way, one gains in flexibility, and may also gain in terms of area and power. In many application areas (e.g., wireless communication), flexibility is introduced even at the cost of area in order to save power. A prominent example of this design choice is running different algorithms to perform the same task with different accuracy and power consumption factors. However, such a concept is not generally explored for cryptographic designs, and to the best of our knowledge, this kind of a hybrid approach has not yet been considered for integrated design of cryptographic accelerators.

Contribution In this paper, we first take up the LTE Stream Ciphers (SNOW 3G and ZUC) as a case study for our idea of integration. There has been a few academic publications towards hardware implementation of the individual ciphers. Especially, Kitsos et al [13] provide us with a high performance ASIC implementation of SNOW 3G and recently Liu et al [14] have published an efficient FPGA based implementation of ZUC. However, the state-of-the-art hardware implementation of both the ciphers come from the commercial domain, especially from Elliptic Technologies Inc. [5–10] and IP Cores Inc. [12], the established brands in the field of hardware security solutions. In each of the above cases, the accelerators for SNOW 3G and ZUC have been developed separately as individual cores, whereas the ciphers are going to be used on the same platform. Moreover, the two ciphers

have significant structural similarities to facilitate an integrated design. This is the driving factor behind our attempt to construct a unified accelerator that would provide higher throughput compared to all existing designs.

We design an integrated high performance accelerator (henceforth called HiPAcc-LTE) for SNOW 3G and ZUC (version 1.5, as in LTE Release 10), targeted towards the 4G mobile broadband market. We merge the two ciphers within a single core by sharing resources among them, thereby reducing the area overhead compared to two independent implementations. HiPAcc-LTE provides *almost twice* the throughput for both the ciphers compared to any existing architecture for the individual algorithms. We also provide the user with the flexibility to choose the ‘area vs. throughput’ trade-off for a customized design. We provide a combined fault detection and protection mechanism in HiPAcc-LTE. In case of SNOW 3G, we provide tolerance against the known fault attack by Debraize and Corbella [3]. For ZUC, as there are no known fault attacks to date, we just leave the room for future fault protection needs.

Furthermore in this paper, we put forward another case study of integrated accelerator design, considering the merger of RC4 and HC-128 at the algorithmic level as well as in implementation. RC4 and HC-128 are both categorized as software stream ciphers. To enhance performance of embedded systems executing such software ciphers, custom instructions or dedicated accelerators are commonly deployed. We propose the implementation of one such dedicated accelerator to execute both RC4 and HC-128. Our integration achieves high throughput with slight area reduction for the accelerator compared to standalone implementations.¹

Long term vision In the long run, this hardware integration approach for cryptographic algorithms will probably result in a flexible core supporting multiple designs including intermediate design points. This strategy will provide the developer to design a unified architecture with optimal performance for a number of cryptographic primitives with similar structural and algorithmic construct. To the end user, the integrated core presents a fast platform to design, validate and benchmark upcoming cipher primitives as well.

Organization The technical content of this paper is organized as follows. Section 2 presents a brief overview of the ciphers SNOW 3G and ZUC. We also present some initial observations regarding the structural similarities and dissimilarities of the two that will help us later in their integration. Section 3 contains the first case study of our hardware integration idea. We restructure the hardware designs of the two ciphers SNOW 3G and ZUC to exploit the similarities to the fullest and design the combined architecture HiPAcc-LTE. Section 4 deals with simulation, testing and synthesis of HiPAcc-LTE. Furthermore, we provide a combined fault detection and correction facility in HiPAcc-LTE, and present the details of our exploration towards increased throughput using loop-unrolling in the ciphers. In Section 5, we discuss the structural

¹By a ‘standalone implementation’, we mean the design and analysis of a cipher when it is considered *not* as a part of the integrated design. From an integrated architecture of ciphers X and Y, say, we obtain a standalone implementation of X by removing all sequential and combinational components that are unique to Y, and are not shared by X. Thereafter, we perform X-specific optimizations on the rest of the architecture to get best performance for cipher X.

similarities of RC4 and HC-128, and construct the combined architecture IntAcc-RCHC by restructuring the pipeline and sharing the resources between the two ciphers. We also present the experimental results for IntAcc-RCHC in this section. Finally, Section 6 concludes the paper by providing a future direction of research oriented around the idea of hardware integration proposed in this paper.

2 Preliminaries

Before the main technical content of this paper, let us first recall the design of SNOW 3G, ZUC and RC4, HC-128.

2.1 Brief overview of SNOW 3G and ZUC

SNOW 3G [21] is an LFSR based stream cipher designed by ETSI-SAGE, largely based on the cipher SNOW 2.0 [4] by Ekdahl and Johansson. The cipher generates a keystream of 32-bit words using an LFSR of size 16 words, that is $16 \times 32 = 512$ bits. The FSM of this design consists of three 32-bit registers which are updated based on two different S-boxes S1, S2. The LFSR update function depends on a couple of field operations (multiplication and division by field element α) and XOR combinations. Alike most stream ciphers, SNOW 3G has two distinct modes of operation. During the initialization mode, the LFSR is initiated using a 128-bit key and a 128-bit initialization variable (IV), and the output of the FSM is XOR-ed with the LFSR update function in the feedback loop for the first 32 iterations. Thereafter, in the keystream generation mode, the output of the FSM is combined with the first LFSR location s_0 to produce the output keystream word. The operation of the cipher in keystream generation mode is shown in Fig. 1.

ZUC [22] is also an LFSR based word oriented stream cipher, designed by the Data Assurance and Communication Security Research Center of the Chinese Academy of Sciences (DACAS). This cipher produces a keystream of 32-bit words, and is executed in two stages (initialization and keystream generation). The LFSR

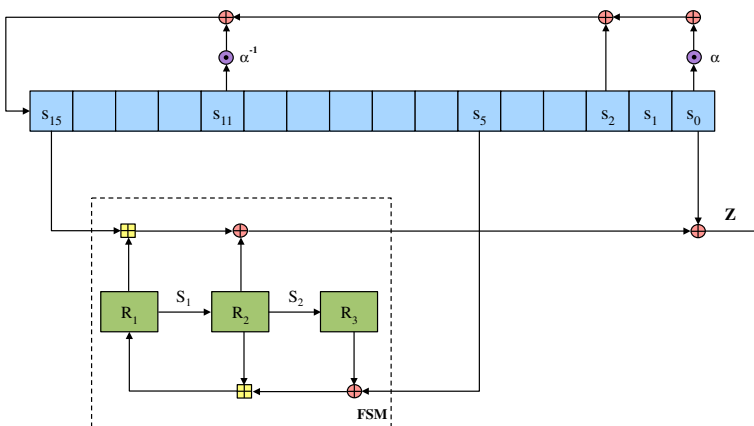


Fig. 1 SNOW 3G cipher in keystream generation mode [21]

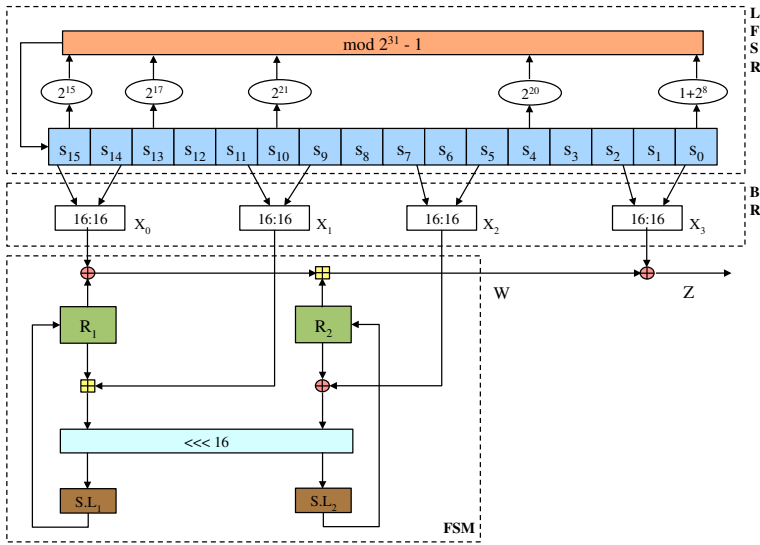


Fig. 2 ZUC cipher in keystream generation mode [22]

for ZUC consists of 16 blocks, each of length 31 bits, and the update function of the LFSR is based on a series of modulo $2^{31} - 1$ (this is a prime) multiplications and additions. The FSM takes as input 32-bit words constructed from the LFSR (through a routine called Bit Reorganization or BR) and outputs a 32 bit word as well. It consists of two 32-bit registers R1 and R2 which are updated using two different linear functions L1, L2 and the same S-box S. The initial state of the LFSR is constructed using a 128 bit key and a 128 bit IV, and during the first 32 iterations, the output of the FSM is added (modulo $2^{31} - 1$ addition after right shift by one place) to the feedback loop for LFSR update. In the keystream generation mode, the output of the FSM is combined with the word X_3 , constructed from the LFSR places s_0 and s_2 , to produce the final output. The keystream generation mode of ZUC is illustrated in Fig. 2.

2.2 Brief overview of RC4 and HC-128

RC4 was allegedly designed by Rivest in 1987, and it is the most widely used commercial cipher to date. The design consists of two major components, the Key Scheduling Algorithm (KSA) and the Pseudo-Random Generation Algorithm (PRGA). The internal state of RC4 contains a permutation of size $N = 256$ words.

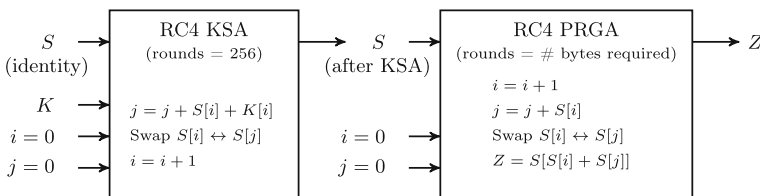


Fig. 3 Key-Scheduling Algo (KSA) and Pseudo-Random Generation Algo (PRGA) of RC4

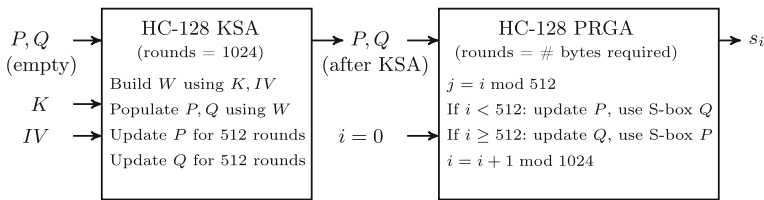


Fig. 4 Key-Scheduling Algo (KSA) and Pseudo-Random Generation Algo (PRGA) of HC-128

The key K is of the same size 256 words as well. However, the original secret key is of length typically between five to 32 words, and is repeated to form the expanded key K . The KSA produces the initial pseudo-random permutation of RC4 by scrambling an identity permutation using key K . The initial permutation S produced by the KSA acts as an input to the next procedure PRGA that generates the output sequence. The RC4 algorithms KSA and PRGA are as shown in Fig. 3 (all additions are modulo 256).

HC-128 [25] is also a state-based stream cipher, designed by Wu and later inducted into the final eSTREAM portfolio [24]. Internally, it consists of two secret tables (P and Q). Each table contains 512 number of 32-bit words. Initially, the 128-bit key and 128-bit IV is used to populate these tables, and then the key-scheduling routine is performed to update the initial states. For each state update one 32-bit word in each table is updated using a non-linear update function. After 1024 steps all elements of the tables have been updated. Thereafter in keystream generation mode, the cipher generates one 32-bit word for each subsequent update step using a 32-bit to 32-bit mapping function. Finally a linear bit-masking function is applied to generate an output word s_i . The two message schedule functions in the hash function SHA-256 [16] are used with the tables P and Q as S-boxes alternately. The main components of operation, KSA and PRGA, are outlined in Fig. 4.

The individual overview of the ciphers SNOW 3G, ZUC and RC4, HC-128 helps us identify the similarities and dissimilarities in their designs, which will lead to their integration, as described in the next sections.

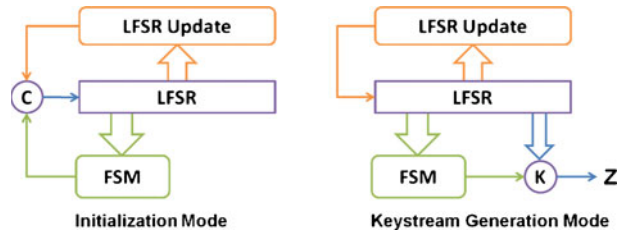
3 HiPAcc-LTE: integrated accelerator for SNOW 3G and ZUC

In this section, we present our main idea behind the architectural integration of SNOW 3G and ZUC. First, we put the two ciphers side by side for a structural comparison in the designs.

3.1 SNOW 3G and ZUC: structural comparison

Similarities The reader may easily spot the inherent structural similarity in the designs of the two ciphers SNOW 3G and ZUC. This is mainly because both ciphers are based on the same principle of combining an LFSR with an FSM, where the LFSR feeds the next state of the FSM. In the initialization mode, the output of the FSM contributes towards the feedback cycle of the LFSR, and in the keystream generation mode, the FSM contributes towards the keystream.

Fig. 5 Top level structure of both SNOW 3G and ZUC



A top level structure for both the ciphers can hence be represented as in Fig. 5. The figure on the left indicates the initialization mode of operation while the figure on the right demonstrates the operation during keystream generation. In Fig. 5, the combination of the LFSR update and the FSM during initialization mode is represented by C, which is either an XOR or a shift and addition modulo $2^{31} - 1$ for SNOW 3G and ZUC respectively. In the keystream generation mode, the combination of the LFSR state with the FSM output is denoted as K, which is an XOR for SNOW 3G and a bit reorganized XOR for ZUC. The operations are individually presented in the previous subsections for the two ciphers. Z represents the output keystream for both the ciphers.

The key point to observe in Fig. 5 is that we have a similar 3-layer structure for both the ciphers SNOW 3G and ZUC. Note that we have not considered Bit Reorganization of ZUC as a special stage, but have taken it as a part of the FSM, thus exhibiting better structural similarity with SNOW 3G.

Dissimilarities As we probe deeper into the individual components of the design, the dissimilarities start appearing one by one. Let us categorize the dissimilarities in the two designs according to the main stages of the ciphers.

1. **LFSR update routine** is fundamentally different for the two ciphers. While SNOW 3G relies on field multiplication/division along with XOR for the LFSR feedback, ZUC employs addition modulo the prime $p = 2^{31} - 1$. Another point to note is that the new updated value s_{15} is required for the next feedback in case of ZUC, whereas SNOW 3G does not have this dependency. This creates a major difference in designing the combined architecture.
2. **The main LFSR** is slightly different for the ciphers as well, although both SNOW 3G and ZUC output 32-bit words. SNOW 3G uses an LFSR of 16 words, each of size 32 bits, whereas ZUC uses an LFSR of 16 words, each of size 31 bits. However, the bit organization stage of ZUC builds 32 bit words from the LFSR towards FSM update and output generation.
3. **FSM operations** of SNOW 3G and ZUC are quite different as well, though they use similar resources. SNOW 3G has three registers R1, R2 and R3 where the updation dependency $R1 \rightarrow R2 \rightarrow R3 \rightarrow R1$ is cyclic with the last edge depending on the LFSR as well. In case of ZUC, there are only two registers R1 and R2. The updation of each depends on its previous state as well as that of the other register. And of course, the LFSR also feeds the state updation process, as in the case of SNOW 3G.

In the next section, we will try to merge the designs of SNOW 3G and ZUC in such a fashion that the similarities are exploited to the maximum extent, and the common

resources are shared. The dissimilarities that we have discussed above will be treated specially for each of the ciphers. We will attempt this merger in three parts, each corresponding to the major structural blocks of the two designs; namely, the main LFSR, the LFSR update function and the FSM.

3.2 Integrating the main LFSR

Recall that the LFSR of SNOW 3G has 16 words of 32 bits each, while that of ZUC has 16 words of 31 bits each. Our first goal is to share this resource among the two ciphers. If we do a naive sharing by putting the 31 bit words of ZUC in the same containers as those for the 32 bit words of SNOW 3G, 1 bit per word is left unused in ZUC. Hence, our first target was to utilize this bit in such a way that reduces the critical path in the overall implementation.

Motivation In Section 4, while discussing the pipeline structure, we will note that the critical path flows through the output channel, that is, through the bit reorganization for s_{15}, s_{14} and s_2, s_0 , and the FSM output of W . In fact, bit reorganization is also required for the FSM register update process. Keeping this in mind, we tried to remove the bit reorganization process from the FSM.

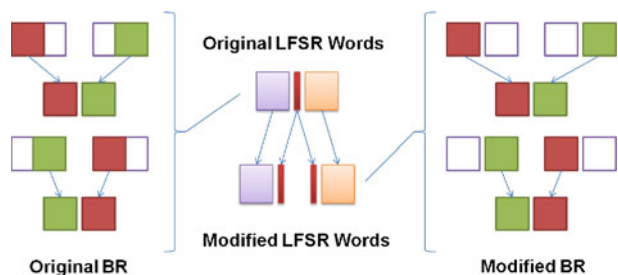
Restructuring the LFSR In this direction, we construct the LFSR as 32 registers of 16 bits each. The 32 bit words for SNOW 3G would be split in halves and stored in the LFSR registers naturally. For ZUC, we split the 31 bit words in ‘top 16 bit’ and ‘bottom 16 bit’ pieces, and store them individually in the 16 bit LFSR registers. The organization of bits is shown in the middle column of Fig. 6, where the two blocks share the center-most bit of the 31 bit original word. Notice that we do not require the bit reorganization any more in the FSM operation, as it reduces to simple *read* from two separate registers in our construction. The modified bit reorganization model is illustrated in Fig. 6.

However, note that the LFSR update function of ZUC uses the 31 bit words for the modulo $2^{31} - 1$ addition. Thus, we have actually moved the bit reorganization stage to the LFSR update stage instead of keeping it in the FSM. The effects of our design choices will be discussed later in Remark 1.

3.3 Integrating the FSM

Although the FSM of the two ciphers do not operate the same way, they share similar physical resources. Thus, our main goal for the integrated design is to share

Fig. 6 Modified bit reorganization for ZUC after LFSR integration



all possible resources between them. Note that the bit reorganization stage is not present in the ZUC FSM any more, due to our LFSR reconstruction.

Register sharing One can straight away spot the registers R1, R2 and R3 for potential sharing. We share R1 and R2 between SNOW 3G and ZUC, while R3 is needed only for the former. If required, R3 can be utilized in ZUC for providing additional buffer towards fault protection, discussed in Section 4.

Sharing the memory During the FSM register update process, both SNOW 3G and ZUC use S-box lookup. In the software version of the ciphers, SNOW 3G [21] uses S_R , S_Q and ZUC [22] uses S_0 , S_1 . However, for efficient hardware implementation of SNOW 3G with memory access, we choose to use the tables $S1_T0$, $S1_T1$, ..., $S2_T3$, as prescribed in the specifications [21]. This saves a lot of computations after the memory read, and hence reduces the critical path to a considerable extent. We store the eight tables in a data memory of size 8 KByte.

For ZUC, however, we can not bypass the lookup to S_0 and S_1 . But one may note that these tables are accessed four times each during the FSM update. So, to parallelize the memory access, we store four copies of each table (thus eight in total) in the same 8 KByte of data memory that we have allocated for SNOW 3G. Note that we are not using the full capacity of the memory in ZUC, as we store 1 byte in each location (as in S_0 and S_1) whereas it is capable of accommodating 4 bytes in each (as in $S1_T0$, $S1_T1$, ..., $S2_T3$).

By duplicating the ZUC tables in the eight distinct memory locations, we have restricted the memory read requests to one call per table in each cycle of FSM. This makes possible the sharing of memory access between SNOW 3G and ZUC as well. We use only a single port to read from each of the tables, and that too is shared between the ciphers for efficient use of resources. This in turn reduces the multiplexer logic and area of the overall architecture.

Pipeline based on memory access Now that we have memory lookup during the FSM update, we partition the pipeline according to it. We simulate the memory by a synchronous SRAM with single-cycle read latency. To optimize the efficiency with an allowance for the latency in memory read, we split the pipeline in two stages, keeping the memory read request and read operations in the middle. The structure of our initial pipeline idea is shown in Fig. 7.

This pipeline is organized around the memory access, where we perform

- the memory read request and LFSR update in Stage 1, and
- the memory read and output computation in Stage 2.

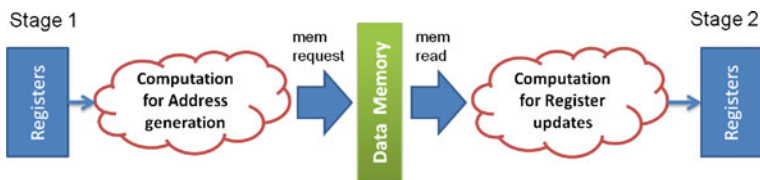


Fig. 7 Pipeline structure based on memory access

For SNOW 3G, the computation for memory address generation is a simple partitioning of R1 and R2 values in bytes. The computation for register update however, requires an XOR after the memory read. In case of ZUC, the computation for address generation is complicated, and depends on the LFSR as well as R1 and R2. However, the computation for register update is a simple concatenation of the values read from memory.

Remark 1 So far, we have made a few design choices in integrating the two ciphers. In a nutshell, the choices provide

- reduction in the critical path by reducing the memory and LFSR read times,
- reduced critical path by moving the bit reorganization away from FSM, and
- an efficient method for combined fault protection in both the ciphers.

The effect of these choices will be reflected in the critical path and fault tolerance mechanism, discussed later in Section 4 of this paper.

Next, we deal with the integration of the most crucial part of the two ciphers: the LFSR update and shift operations. The final structure of the pipeline will evolve during this phase as we deal with the intricate details in the design.

3.4 Integrating the LFSR update function

The LFSR update function is primarily different for the two ciphers. The only thing in common is the logic for LFSR update during initialization, and this poses a big problem with our earlier pipeline idea based on memory access (Fig. 7).

Pipeline restructuring for key initialization In the initialization mode of the two ciphers, the FSM output W is fed back to the LFSR update logic. The update of s_{15} takes place based on this feedback, and in turn, this controls the next output of the FSM (note that W depends on R1, R2 and s_{15} in both ciphers). This is not a problem in the keystream mode as the LFSR update path is independent of the output of FSM. However, during initialization, it creates a combinational loop from Stages 2 to 1 in our earlier pipeline organization (Fig. 7). This combinational loop in memory access due to dependencies prohibits us from keeping the memory access and memory read in two different stages of the pipeline. Thus, we design a new structure as follows:

- Stage 1: Initial computation for memory access and LFSR shift.
- Stage 2: Memory read, LFSR update and subsequent memory read request.

This new pipeline structure allows us to resolve the memory access dependencies within a single stage and the independent shift of the LFSR occurs in the other. Now, the main goal is to orient the LFSR update logic around this pipeline structure, or to redesign the pipeline according to the LFSR update function.

Pipeline organization for LFSR update The LFSR update logic of SNOW 3G is easier to deal with. The update depends upon the LFSR positions s_0 , s_2 and s_{11} , and also on the FSM output W during key initialization. A part of s_0 and s_{11}

each undergoes a field operation (MUL_α and DIV_α respectively), and the other part gets XOR-ed thereafter. To reduce the combinational logic of realizing the field operations, two lookup tables are prescribed in the specifications [21]. For an efficient implementation in hardware, we follow this idea and store the two tables MUL_α and DIV_α in two 1 KByte memory locations. These are also read-only memories with single-cycle read latency. Now, we can fit the update routine for SNOW 3G within the two stage pipeline proposed earlier.

- Stage 1: Precompute the simple XOR involving s_0, s_2 and s_{11} , and generate the addresses for memory read requests to tables MUL_α and DIV_α .
- Stage 2: Perform memory read and XOR with the previous XOR-ed values to complete the LFSR feedback path, run the FSM and complete the LFSR update of s_{15} depending on W .

Note that this pipeline structure works both for initialization as well as keystream generation, as it takes into account all possible values required for the LFSR update. Thus, in terms of SNOW 3G, we stick to our 2-stage pipeline.

In case of ZUC however, the LFSR update logic is quite complicated. This is mostly because of the additions modulo the prime $p = 2^{31} - 1$. Liu et al [14] had proposed a single adder implementation of this addition modulo prime, and this logic has also been included in the specifications [22]. We use the same for our hardware, at least at this initial phase. In the same line, we first try a 5-stage pipeline, similar to the one proposed in [14] for LFSR update of ZUC.

The initial idea for 5-stage pipeline is shown as Pipeline 1 in Fig. 8. All the adders are modulo prime, similar to the ones in [14], and the variables a, b, c, d, e, f

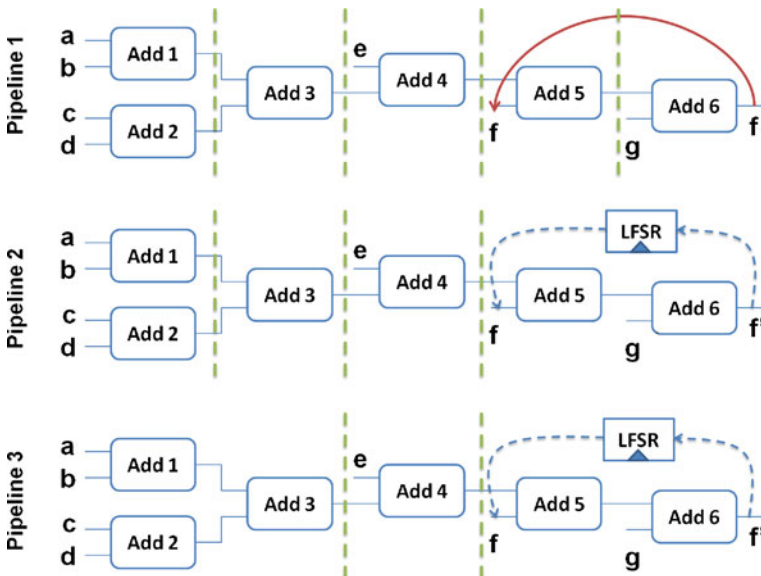


Fig. 8 Pipeline structure reorganization for LFSR update of ZUC

represent $s_0, 2^8s_0, 2^{20}s_4, 2^{21}s_{10}, 2^{17}s_{13}, 2^{15}s_{15}$ (modulo $p = 2^{31} - 1$) respectively. Variable g denotes the FSM output W , which is added with the cumulative LFSR feedback, and is then fed back to s_{15} in the LFSR itself.

However, Pipeline 1 creates a combinational loop between Stages 5 and 4 in the key initialization phase. The final output in Stage 5 of the addition pipeline has to be fed back to s_{15} that controls the input f in Stage 4. This loop is shown by the curvy solid line in Fig. 8, and it occurs due to mutual dependency of FSM and LFSR update during initialization. The authors of [14] also observed this dependency, and they proposed the 32 rounds of key initialization to be run in software in order to achieve one-word-per-cycle using their structure.

Our challenge was to integrate this phase into the hardware without losing the throughput. The main motivation is to restrain the use of an external aide for the initialization mode. There are two direct ways of resolving this issue:

1. Allow a bypass logic for the f value across the stages
2. Restructure the pipeline to merge the last two stages

We choose the second one and reorganize the pipeline. As the dependency discussed so far occurs in between the last two stages of the pipeline, we merge those to resolve the inter-stage combinational loop. In this case, the output f' of this stage is written into the s_{15} location of the LFSR, and read back as f at the next iteration. This is shown as Pipeline 2 in Fig. 8.

The reader may note that we have two adders (modulo prime p) in series at the last stage of Pipeline 2 (Fig. 8). So, we can put two adders in any other stage as well, without affecting the critical path. We decide to merge Stages 1 and 2 to have two adders in parallel followed by an adder in series in the first stage. This does not increase the critical path, which still lies in the last stage due to the two adders and some associated combinational logic. The final structure of the LFSR update pipeline for ZUC is shown in Fig. 8 as Pipeline 3. In the next section, we design the integrated pipeline structure combining all components.

3.5 Final design of the pipeline

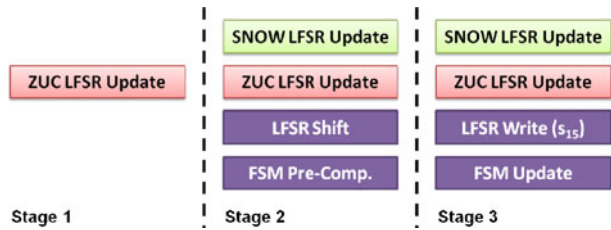
In this section, we present the final pipeline structure for the integrated architecture. In the previous sections, we have already partitioned the components into pipeline stages as follows.

- FSM: Two stages—initial computations for address generation in the first stage, and memory access and related computations in the second stage.
- LFSR Movement: Two stages—shift in first stage and s_{15} write in second.
- LFSR Update: Two stages for SNOW 3G and three stages for ZUC.

Here, we combine all three components of SNOW 3G and ZUC and design the final pipeline for our proposed hardware implementation, as shown in Fig. 9.

The stages of SNOW 3G and ZUC are different only in case of the LFSR update routine, and we show these separately in the figure. The pipeline behavior of the LFSR shift and write operations, as well the FSM precomputation and update routines are almost same for both the ciphers, and hence we show single instances

Fig. 9 Final 3-stage pipeline structure for the integrated design



of these in Fig. 9. In the next section, we discuss the practical issues with the final ASIC implementation of our integrated hardware.

4 ASIC implementation of HiPAcc-LTE

In this work, we utilized the hardware generation environment and simulation framework from LISA, the Language for Instruction-Set Architectures, for designing the accelerator. The complete automatic generation environment is commercially available via Synopsys Processor Designer [23]. The accelerator in our case is designed as a state machine. This allowed fast exploration of design alternatives and ease of high level modeling for making pipelining and resource organization decisions. The language allows full control over minute design decisions and preserves the overall structural organization neatly in the generated hardware description [18]. This is especially important for verifying the design costs (area, timing) and accordingly modifying the design at high level. Such a capability of strong designer interaction with the tool during high level synthesis is not common among automatic C to HDL flows [17], thereby forcing designers to go through time consuming and error prone low-level design iterations.

The gate-level synthesis was carried out using Synopsys Design Compiler Version D-2010.03-SP4, using topographical mode for a 65 nm target technology library. The area results are reported using equivalent 2-input NAND gates. The total lines of LISA code for our best implementation is 1131, while the total lines of auto-generated HDL code is 13440 for the same design. The modeling, implementation, optimization and tests were completed over a span of two weeks.

In this section, we first discuss the issues with the *critical path* in our design, and the optimizations thereof. This will be followed by a set of detailed implementation results and comparisons with the existing designs.

4.1 Critical path

After the initial synthesis of our design using LISA modeling language, we identified the critical path to occur in the key initialization phase of ZUC. Figure 10 depicts the critical path using the curvy dashed line. To understand the individual components in the critical path, let us first associate the pieces in Fig. 10 to the original initialization routine of ZUC, as described in its specification [22]. Later, we shall perform a set of optimizations on the design.

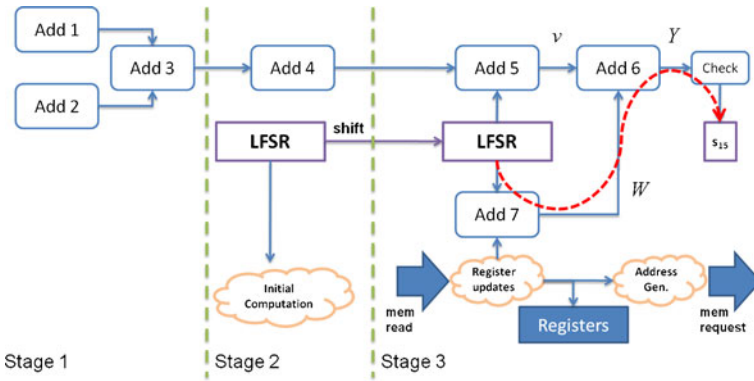


Fig. 10 Critical path in the key initialization of ZUC (curvy dashed line)

ZUC key initialization routine The following is the key initialization routine of ZUC, as per our notation and pipeline orientation. Note that the operation is the same as in the `LFSRWithInitialisationMode()` function of [22].

`LFSR_Key_Initialization (W)`

1. $v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + 2^8s_0 + s_0 \pmod{2^{31} - 1}$
2. $Y = v + (W \gg 1) \pmod{2^{31} - 1}$
3. If $Y = 0$, then set $Y = 2^{31} - 1$
4. Write Y to location s_{15} of the LFSR

In Fig. 10, the first five adders Add one to Add five are part of the general LFSR feedback loop in ZUC, and they compute the value

$$v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + 2^8s_0 + s_0 \pmod{2^{31} - 1}.$$

The LFSR is also accessed to run the FSM and the adder Add seven at the bottom of Stage 3 computes the FSM output $W = (X_0 \oplus R_1) + R_2$, where this addition is a normal 32-bit addition. The special operation in LFSR update of ZUC in its initialization mode is to compute $Y = v + (W \gg 1) \pmod{2^{31} + 1}$, realized by the adder Add six on the top layer of Stage 3. If this sum $Y = 0$, it is replaced by $Y = 2^{31} - 1$ in the ‘Check’ module of Fig. 10. Finally, this 31 bit value Y is written to s_{15} of the LFSR, thus completing the LFSR update loop. The critical path, as shown by the curvy dashed line in Fig. 10, is as follows:

LFSR Read → 32-bit Add → Modulo Add → Check → LFSR Write

In this section, we try all possible optimizations to reduce the critical path.

LFSR read optimization At first, we implemented the LFSR as a register array. However, different locations of the LFSR are accessed at different stages of the pipeline we have designed, and the LFSR read will be faster if we allow the individual LFSR cells to be placed independently in the stages. This motivated us to implement the LFSR as 32 distinct registers of size 16 bits each. Furthermore, we shadowed the last two locations, i.e., s_{15} of the LFSR, so that it can be read instantaneously from both Stages 4 and 5. This led to a reduction in the critical path. Though this

optimization is targeted towards physical synthesis, the gate-level synthesis results indicated strong improvement as well.

Modulo p adder optimization Initially, we designed the modulo $p = 2^{31} - 1$ adder as prescribed in [14]. This looks like the circuit on the left of Fig. 11. However, one may bypass the multiplexer (MUX) by simply incrementing the sum by the value of the carry bit. That is, if the carry bit is 1, the sum gets incremented by 1, and it remains the same otherwise. The modified design (right side of Fig. 11) slightly reduces the critical path and we replace all the modulo p adders in our design (except for Add 6) by this modified circuit.

Check optimization The ‘Check’ block in the critical path actually has two checks in series; one due to Add six where the increment is based on the carry bit, and the second check is for the $Y = 0$ situation. We try to optimize as follows.

- Carry = 0: We just require to check if $Y = 0$. If so, set $Y = 2^{31} - 1$.
- Carry = 1: We just require to set $Y = Y + 1$ without any further checks.

The first case is obvious, as the sum would remain unchanged if the carry is 0. In the second case, note that the inputs v and ($W \gg 1$) to Add six are both less than or equal to $2^{31} - 1$. Thus, the sum Y is bounded from above by $2^{32} - 2$. Even if the carry is 1, the incremented value of sum will be bounded from above by $2^{32} - 1$, which can never have the lower 31 bits all equal to 0. Thus, we do not even require the ‘Check’ block in this situation. This optimization simplifies the logic and reduces the critical path considerably.

4.2 Performance results

After performing all the optimizations discussed in the previous section, we still find the critical path flowing through the same components. We proceed for final synthesis and performance results based on the current state of the design.

Table 1 presents all the architecture design points for HiPAcc-LTE that we have implemented using the 65 nm technology. The area-time chart for the design points of HiPAcc-LTE is shown in Fig. 12.

The maximum frequency we could achieve is 1090 MHz, which corresponds to a critical path length of approximately 0.92 ns. This provides us with a net throughput of 34.88 Gbps, with 1 keystream word per cycle. The total area is about 17 KGates NAND equivalent and 10 KByte of data memory is required.

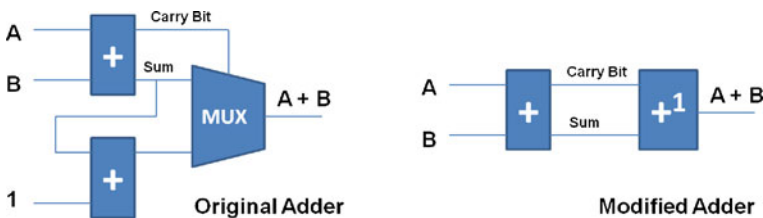


Fig. 11 Modulo p adder optimization for ZUC

Table 1 Synthesis results for HiPAcc-LTE with 10 KByte memory

Frequency (MHz)	Area (equivalent NAND gates)		
	Total	Sequential	Combinational
200	11699	5540	6159
500	13089	5540	7549
800	14102	5541	8561
1000	15696	5541	10155
1050	16055	5554	10501
1090	16886	5568	11318

Experiments with reduced data memory In the original HiPAcc-LTE design as above, the static data for S-box and field operations have been stored in external data memory. While SNOW 3G utilizes the complete 10 KByte memory, ZUC requires only about 2 KByte of the allocated space. This motivated us to experiment with an alternate design that requires less data memory.

In the alternate design, we use S-box tables S_R , S_Q for SNOW 3G [21] instead of the tables $S1_T0$, $S1_T1$, ..., $S2_T3$, as in the previous case. During the sharing of memory, the ZUC tables S_0 , S_1 fit exactly in the space for S_R , S_Q as they are of the same size, 256 bytes each. There are exactly four calls to each table per cycle, and we store two copies of each table in dual-port RAMs to get optimum throughput. This amounts to a data memory of $2 \times (256 + 256)$ bytes = 1 KByte. The MUL_{α} and DIV_{α} tables (size 1 KByte each) in case of SNOW 3G could not be avoided due to the complicated combinational logic involved in these field operations. The total data memory for this alternate design sums up to 3 KByte, and the details for all design points are presented in Table 2.

This alternate design retains the maximum frequency of 1090 MHz, which provides us with a net throughput of 34.88 Gbps, with 1 word per cycle. The area figure is still about 17 KGates NAND equivalent, but only 3 KByte of external data memory is required. It is interesting to note that the combinational area remained almost similar even after introducing the computations for S-boxes. This is possibly due

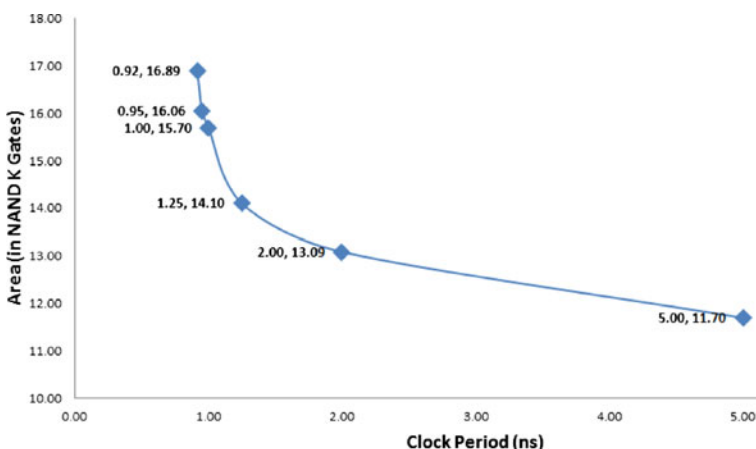
**Fig. 12** Area-Time chart for HiPAcc-LTE (10 KByte memory) using 65 nm technology

Table 2 Synthesis results for alternate design of HiPAcc-LTE with 3 KByte memory

Frequency (MHz)	Area (equivalent NAND gates)		
	Total	Sequential	Combinational
200	10519	5548	4971
500	13090	5540	7550
800	14103	5541	8562
1000	15696	5541	10155
1090	16887	5568	11319

to the availability of high-speed, area-efficient library cells in our target technology library and efficient design style.

With this alternate design of HiPAcc-LTE having 3 KByte of memory, the performance of the individual ciphers SNOW 3G and ZUC are also tested in standalone mode. The synthesis results in this direction are presented in Table 3.

4.3 Exploration of storage implementation

For the physical implementation of the storage, a number of alternatives are explored. The choices are primarily limited by the constraints like read-only configuration, number of access ports. For FPGA-based designs, while it is commonplace to exploit available RAM blocks, storage must be designed carefully for ASIC implementation. We utilized Faraday Memory Compiler with 65 nm technology library for exploring dual-port block RAMs and synchronous ROMs. While block RAMs can be utilized for both SNOW and ZUC execution, the ROM is not re-programmable and therefore, must hold the complete storage for both the algorithms.

For SNOW 3G, the RAM requirement is higher due to additional tables for MUL_{α} and DIV_{α} computation. The synthesis results show approximately 43 KGates for SNOW 3G and 26.8 KGates for ZUC. The memory access time is slower than the combinational path of the logical operations thereby, supporting the highest achievable frequency.

For synchronous ROM, Faraday Memory Compiler supports a minimum size of 4096 bits with 1 read port, which is more than our requirement. For supporting the parallel computation both SNOW 3G and ZUC requires 8 2048-bit ROM with 8-bit word alignment and 1 read port access. With this forced redundancy of double data capacity with limited port access, the ROM synthesizes to approximately 23.12 KGates for ZUC. Similar area i.e. total of 46.24 KGates will be required for SNOW 3G and ZUC even without storing the 2KB tables for MUL_{α} and DIV_{α}

Table 3 Synthesis results for standalone mode in HiPAcc-LTE with 3 KByte memory

Cipher	Frequency (MHz)	Area (equivalent NAND gates)		
		Total	Sequential	Combinational
SNOW 3G	500	6867	5061	1807
	1000	7033	5062	1971
ZUC	500	9555	4798	4757
	1000	11412	4811	6601

computation. Clearly, with port access restrictions synchronous ROM is not a good choice compared to RAM.

We finally attempted to manually code the tables in a switch-case statement and directly synthesize that as hard macro for both SNOW 3G and ZUC. This resulted in much less area compared to the RAM. The results are summarized in Table 4. It must be noted that due to read-only nature of the hard macro, both SNOW 3G and ZUC tables are encoded in the combined design. This also requires multiplexing between alternative tables according to the actual algorithm being executed. As a result, the throughput achievable in the combined design with hard macro is slightly less compared to the design implementing ZUC standalone. A nice advantage of storage implementation with hard macro is that it is less susceptible to physical attacks like memory readout or fault injection. The hard macro is realized within the combinational blocks, whereas RAM or ROM structures maintain a clear separation from the logic and are easier to spot in a physical layout.

4.4 Comparison with existing designs

To put the performance of HiPAcc-LTE into perspective, we compare it with the state-of-the-art architectures available in academia and the commercial sector.

Comparison with academic literature In the domain of published academic results, we could not find an ASIC implementation of ZUC, and neither could we find a 65 nm technology implementation of SNOW 3G. The only hardware realizations for ZUC have been done in FPGA [14] so far. Thus, we could not compare HiPAcc-LTE to any academic results in terms of ZUC.

In case of SNOW 3G, the best academic publication is [13] that uses 130 nm technology. To compare with this result, we synthesized our proposed design (with 10 KByte data memory) in 130 nm, and the comparison is as follows.

- SNOW 3G of [13]: 7.97 Gbps with 249 MHz max. freq. and 25 KGates area
- Our HiPAcc-LTE: 24.0 Gbps with 750 MHz max. freq. and 18 KGates area

Table 4 Comparison of HiPAcc-LTE with existing 65 nm commercial designs

Performance of Commercial Designs					
Cipher	Name of design	Designer	Max. freq. (MHz)	Throughput (Gbps)	Total area (KGates)
SNOW 3G	SNOW3G1 [12]	IP Cores Inc.	943	7.5	8.9
ZUC	CLP-410 [8]	Elliptic Tech.	500	–	10–13
Performance of HiPAcc-LTE					
Cipher	Mode of design for static tables	Memory (KGates)	Frequency (MHz)	Throughput (Gbps)	Total area (KGates)
SNOW 3G		43.0	1000	32.0	50.0
ZUC	3 KByte memory	26.8	1000	32.0	38.2
Both		43.0	1090	34.9	59.9
SNOW 3G		–	1650	52.8	18.1
ZUC	Hard macro	–	920	29.4	20.6
Both		–	900	28.8	27.4

Both designs use about 10 KByte of external data memory for look-up tables. It is clear that we achieve surprisingly better throughput from HiPAcc-LTE due to our careful pipeline design. Our integrated implementation for both the LTE stream ciphers even outperforms the single standalone core in terms of area.

Comparison with commercial designs In the commercial arena, the best architectures available for SNOW 3G and ZUC are from IP Cores Inc. [12] and Elliptic Tech Inc. [8] respectively. Both provide standalone solutions for the individual stream ciphers and match our technology of 65 nm. One tricky issue in the comparison is the area required for the memory. It is not always clear from the product white-paper whether additional memories have been used.

For the sake of fairness, we first compare our designs using 3 KB memory with existing standalone ZUC and SNOW 3G implementations. The memory is synthesized with Faraday Memory Compiler in 65 nm technology node. Further, we replace the S-Box SRAM implementations with hard macros in the RTL design and obtained the gate-level synthesis results. From the commercial designs, the designs with best performance claims in 65 nm technology node are selected. We provide the detailed comparison and analysis in Table 4.

Area comparison: Around an operating frequency of 200–500 MHz, if one uses the two best cores separately, the combined area comes around 18–20 KGates. HiPAcc-LTE synthesizes within 16–18 KGates in this frequency zone (using hard macros), hence offering about 10 % reduction in area. Even with this reduced area figure, HiPAcc-LTE offers the same throughput as CLP-410 [8] and *more than double* throughput compared to SNOW3G1 [12].

Throughput comparison: The best throughput (1 word/cycle) is provided by the CLP-410 ZUC core from Elliptic Tech. However, they just quote a figure of 6 Gbps for 200 MHz [8]. A simple scaling to their maximum frequency of 500 MHz would translate this to an estimate of 15 Gbps. Even in this case, the throughput 29.4 Gbps of HiPAcc-LTE (in hard macro design) is *almost double* compared to any of the commercial standalone implementations of the ciphers.

For a very rough estimate, if one wants to achieve a comparable throughput (approx. 30 Gbps) using the existing standalone modules, then four parallel blocks of SNOW3G1 [12] and two parallel blocks of CLP-410 [8] would be required. This amounts to a total area of roughly 56–62 KGates, while HiPAcc-LTE achieves the same using only 27.4 KGates (at least 51 % reduction) for the hard macro based design. For the sake of fairness, one may also note that we have a comparable area figure of 59.9 KGates for an even higher throughput (34.9 Gbps) using 3 KByte of external data memory. If the extreme throughput is not required for communication purpose, it may facilitate a scaling in frequency/voltage for reduced power consumption.

4.5 Power consumption/dissipation analysis

Power consumption and dissipation are serious design concerns in embedded systems, in particular for the cryptographic devices. Here we present a power estimation of different design points, i.e., the standalone SNOW 3G implementation, standalone ZUC implementation, and the combined design HiPAcc-LTE running individual

Table 5 Power estimation results for HiPAcc-LTE with hard macro storage

Cipher	Frequency (MHz)	Power (mW)	Energy (picoJoule/byte)
SNOW 3G standalone	1650	14.41	2.19
ZUC standalone	920	18.7	5.09
HiPAcc-LTE (SNOW 3G)	900	17.32	4.81
HiPAcc-LTE (ZUC)	900	16.83	4.67

applications. The operating condition of the target 65 nm technology library is set at the best case scenario with a global operating voltage of 1.32 V and temperature -40°C . The power consumption is estimated on a gate-level netlist by back-annotating the switching activity and using Synopsys Power Compiler tool. The results are presented in Table 5.

From Table 5 it can be observed that the standalone SNOW 3G implementation is much more energy-efficient due to its significantly high clock frequency compared to the standalone ZUC implementation. Higher power consumption for ZUC is due to its higher computational complexity. For the combined architecture HiPAcc-LTE, executing SNOW 3G is comparable in terms of energy-efficiency to executing ZUC. The combined architecture is slightly more energy-efficient compared to the standalone ZUC architecture. This is possibly due to the efficient technology mapping for ZUC-specific data-path in the combined architecture.

Typical power optimizations like clock gating and operand isolation for sequential and combinational logic respectively are attempted. This can be easily done by modifying the synthesis script to search for power optimization options based on the annotated switching activity. A minimum bit-width of six and maximum fan-out of 64 is set for clock gating via the synthesis option *set_clock_gating_style*. Adaptive mode of operand isolation is activated via the inbuilt synthesis option *set_operand_isolation_style*. For none of the architectures, clock gating or operand isolation could lower the power consumption. This is understandable from the fact that all the computing blocks and sequential storage cells are active in every cycle. Only a few registers, reserved for the computation of ZUC, are left out during the execution of SNOW on the combined architecture. Clearly the clock gating logic does contribute more than the power it potentially saves. Similarly for the operand isolation, the addition operations are shared between SNOW 3G and ZUC data-path in the combined architecture. This leaves no room for improving power via operand isolation.

4.6 Towards increased throughput

A common technique for increasing throughput in LFSR-based stream ciphers is to unroll and interleave multiple iterations. Experiments in this direction have been attempted on HiPAcc-LTE, and we report the results here.

Unrolled SNOW 3G: Figure 13 shows the structure of SNOW 3G when two consecutive iterations are executed simultaneously for the keystream generation. In the initialization mode, the output is used for loading the word in LFSR. In our proposed implementation, the leftmost word s_{15} is generated in the final pipeline

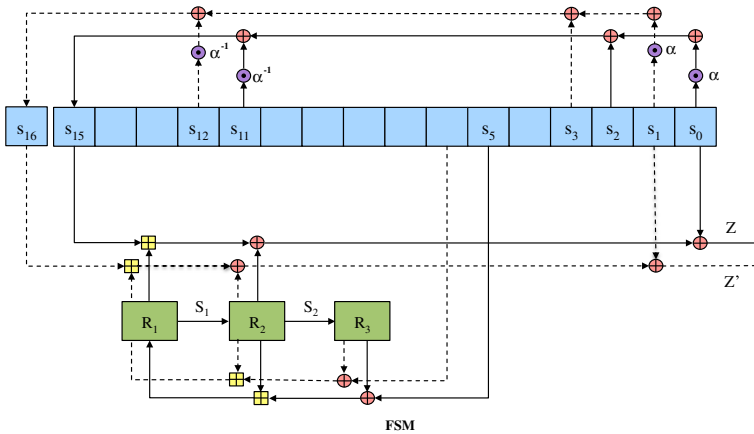


Fig. 13 Two iterations of SNOW 3G in keystream generation mode

stage to efficiently distribute the critical path. The final pipeline stage computes the output Z based on the current values of $R1$, $R2$ and $R3$. Furthermore, the addresses for accessing these tables are generated. For the unrolled structure, an additional word, s_{16} is needed. Since s_{16} is to be produced in the same cycle, $R1$, $R2$ and $R3$ needs to be accessed first for s_{15} and then for s_{16} . This is only possible with a *asynchronous* storage structure for the tables. The hard macro storage style serves this purpose. However, the overhead in timing is significant compared to single-iteration SNOW implementation. Also, unrolling does not offer any area savings since, all the computing blocks as well as the storage macros need to be duplicated.

Unrolled ZUC: The structure of ZUC stream cipher with two interleaved iterations is presented in Fig. 14. Due to the LFSR populating via output during initialization, exactly same issues as for SNOW is also present in ZUC. Moreover, the leftmost LFSR word in ZUC contains a self-feedback loop worsening the clock timing. We distributed the modulo adder operations of LFSR feedback loop in three pipeline stages. However, the long timing-critical path via self-feedback loop of s_{15} , s_{16} remains in the final pipeline stage. This fact, in addition to the timing-critical path via the $R1$ and $R2$ tables, led to poor timing results for ZUC.

Performance: Synthesis results for standalone SNOW 3G and ZUC implementations, after unrolling and interleaving two rounds, are presented in Table 6. In particular, for ZUC, the highest achievable frequency reduced more than two times. As a result, no throughput increase in ZUC is offered. Significant throughput increase is achievable for SNOW. However, a comparison with Table 4 reveals that Area-Time product for the unrolled structure is worse compared to original HiPAcc-LTE design.

4.7 Fault detection and protection in HiPAcc-LTE

To date, no significant fault attack has been mounted on ZUC, and the best fault attack against SNOW 3G has been reported in [3]. In HiPAcc-LTE, we provide

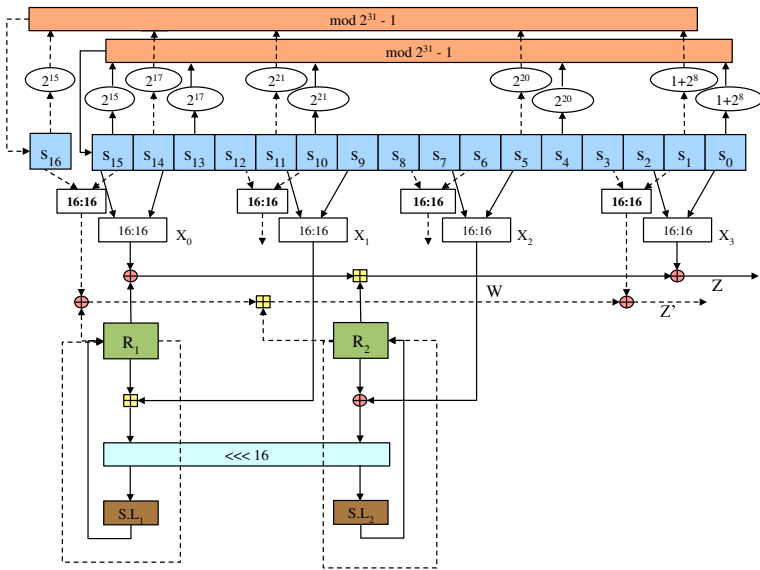


Fig. 14 Two iterations of ZUC cipher in keystream generation mode

detection and protection against this fault attack of SNOW 3G, and provide room for tolerance against future fault attacks on ZUC, if any.

In [3], the authors themselves propose a method to prevent their fault attack in hardware. They have shown that if one shadows the five LFSR locations s_0, s_1, s_2, s_3, s_4 continuously, the attack becomes impossible [3, Section X]. In the hardware implementation of HiPacc-LTE, we have additionally implemented this shadowing mechanism as well. This is realized by keeping a buffer register of $5 \times 32 = 160$ bits which continuously shadows the five LFSR locations by shifting the array by one word in sync with the LFSR shift, and by recording the value of s_5 in the array during Stage 2 of the pipeline (note that this becomes the shadowed value of s_4 in Stage 3). A fault is detected in this locations by comparing the values in the LFSR with the shadowed values from the buffer array, and the keystream byte is not produced if a fault is detected.

The fault tolerance mechanism does not affect the critical path, and HiPacc-LTE still achieves a maximum frequency of 1090 MHz. However, the area figures rise slightly, as expected. Compared to the original HiPacc-LTE, the new area figures increase by approximately 1.5 KGates at 1090 MHz in the 65 nm technology, when the design is implemented using external data memory. The design automatically provides a mechanism for 160 bit shadowing for ZUC, if required, and this is where our earlier design choices for resource sharing (as mentioned in Remark 1) prove to be effective.

Table 6 Performances of two-rounds unrolled ciphers SNOW 3G and ZUC

Cipher	Frequency (MHz)	Total area (KGates)	Throughput (Gbps)
SNOW 3G	1000	29.63	64
ZUC	425	27.69	27.2

So far, we have discussed the hardware integration of the 3GPP LTE-Advanced stream cipher SNOW 3G and ZUC. In the next section, we focus on the integration of RC4 and HC-128, the two most prominent stream ciphers of today.

5 IntAcc-RCHC: integrated accelerator for RC4 and HC-128

Though both HC-128 and RC4 are software stream ciphers, yet it is worthwhile to investigate their hardware accelerator design for several reasons. First, for the system designer the software or hardware design notion is not fixed. A custom instruction-set for AES, when implemented in a general-purpose processor [11], actually uses dedicated hardware components inside the processor pipeline. There, the underlying implementation for a software cipher primitive is, actually hardware. Second, custom accelerators are increasingly found in modern System-on-Chips (SoCs). The increasing area capacity of a single chip can accommodate more and more components. The key point there is to obtain runtime efficiency and thereby reduce energy consumption. Area-driven design is replaced by energy-driven design paradigm. To reduce energy, dedicated accelerators serve better purpose than a general-purpose processor.

Recall the structure of RC4 and HC-128 from Section 2. Given the structural similarity of HC-128 and RC4, there is a possibility of a shared design, where either of the ciphers can be run by dedicated configurations. From the algorithmic viewpoint, following resources can be shared readily between the two ciphers.

- *Counter*: Both HC-128 and RC4 uses iterators, which can be easily shared.
- *Temporal Distribution of Operations*: Similar to HiPAcc-LTE, we attempt to obtain similar pipeline structures for the two algorithms while distributing the operations in different stages of the pipeline. We need to also respect the SRAM port access restrictions.
- *Storage*: Given the large storage size of HC-128 and both the read and write operation requirement, block SRAM is only possible option. For HC-128, the arrays P , Q and W needs to be stored in SRAM with total 9 KB memory requirement. RC4 requires much less memory for its S and K arrays. However, we show later that by re-using the complete structure of HC-128, we can actually benefit for RC4 design.

Now that we have identified potential spots for integration, we start by constructing the pipeline for the combined design IntAcc-RCHC. As HC-128 contains computationally more demanding key generation kernel as well as larger storage requirements, we focus on the HC-128 structure first.

5.1 Restructuring HC-128 pipeline for integration

Memory access: In the PRGA stage of the cipher, there is a high number of storage accesses for the arrays P and Q . Thus we use a 2-read, 1-write dual-port SRAM block for P , Q arrays. The number of accesses directly influences the pipeline design for the combinational path and hence, the throughput. The storage accesses of key stream generation algorithm are distributed in the 4-stage pipeline structure (as shown in

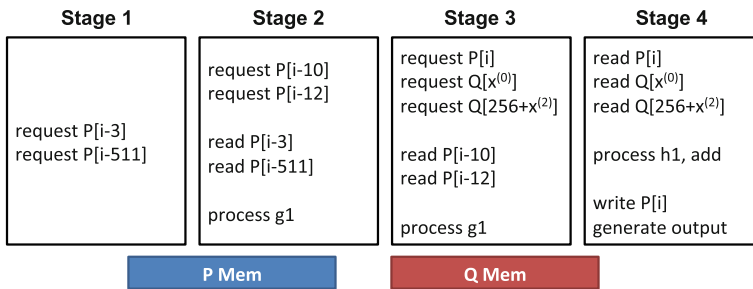


Fig. 15 HC-128 pipeline (PRGA) for integrated design IntAcc-RCHC

Fig. 15) in a manner that the RAM access restrictions are not violated if only one pipeline stage is active at any given cycle.

Computation: Efficient distribution of computation in the pipeline stages is important for achieving a low critical path, i.e., a high clock frequency. To that effect, the computation is triggered from the innermost kernel of functions $g1$ or $g2$, i.e., the rotate operation followed by the XOR is computed as soon as the data is available in pipeline Stage 2. In Stage 3, further processing of $g1$ is done and the addresses for accessing Q array is computed. In the same stage, the read requests for Q array are placed. In the final stage (Stage 4), the remaining computations for key stream generation and updating the P array are performed. Subsequently, a write request to P array is made.

Throughput: Note that, because of this write request, a new iteration of key stream generation cannot be started until the next cycle. This means, every iteration of key stream generation occupies the complete pipeline for 4 cycles, thereby providing a key stream generation speed of 1 word per 4 cycles or 1 byte/cycle.

Key initialization: The initialization module of HC-128 is also included in the design. The initialization part of HC-128 is distributed into four phases. The first three phases perform a given set of operations using P , Q and auxiliary array W . The last phase essentially runs the key stream generation step 1024 times, where instead of generating the final key stream, the arrays are updated with the final result at each step.

5.2 Restructuring RC4 pipeline for integration

Using dual-port SRAM, 1 byte per 3 cycle performance from an RC4 structure is suggested in the literature [15]. It is also suggested that increasing the number of ports can increase the throughput. However, with block RAMs, the number of ports are almost always limited by 2. Few commercial offerings with 4-port SRAMs are available though at a high area and timing overhead compared to the dual-port SRAM.

Memory access: We propose a novel pipeline distribution (as shown in Fig. 16) of the data-path and re-use both P and Q memories of HC-128 to obtain 1 byte

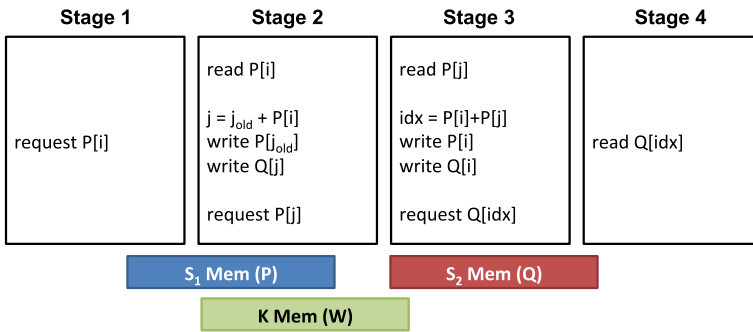


Fig. 16 RC4 pipeline (PRGA) for integrated design IntAcc-RCHC

per 2 cycle throughput. In our design, both P and Q memories store the S-box. This requires write update to the both the memories when the $S[i]$ and $S[j]$ values are exchanged. For reading the data for swap or output generation, both P and Q memories are used. It can be noted that by inserting a dummy operation between every two successive keystream generation calls, either Stages 1 and 3 or Stages 2 and 4 are simultaneously active. This ensures that the memory port access restrictions are not violated.

Computation: In order to maintain the data coherency, bypass paths are suggested in the RAM structure of [15]. Noting that it affects critical path, we avoid any bypass logic in this configuration. We use multiple copies of the same, i.e., replicated memory system. In RC4, this leads to two situations where data access conflicts might result.

- First, $P[j]$ write is requested before $P[j]$ is read in Stage 3. Since both read and write are synchronous, the read and write accesses will actually take place simultaneously. To ensure correct operation, P memory should be configured to no access conflict. This is achieved by writing the $P[j_{old}]$. This can lead to wrong value of $P[i]$ in the following iteration, which can be checked by comparing i and j_{old} .
- Second situation might arise in Stage 4 if idx is equal to i and $P[i]$ or $Q[i]$ write needs to occur before the read. This can be easily checked by comparing i and idx values in pipeline Stage 3.

The pipeline described above for RC4 leads to a seamless integration of the cipher within an existing 4-stage pipeline implementation of HC-128. All the memory resources of HC-128 are shared by RC4, thus reducing the area overhead compared to two individual instances of the ciphers. We study the performance of our combined design IntAcc-RCHC in the next section.

5.3 Implementation and performance of IntAcc-RCHC

The experimental setup for IntAcc-RCHC is the same as that for HiPAcc-LTE, as described in Section 4. The gate-level synthesis was carried out using Synopsys Design Compiler Version D-2010.03-SP4, using topographical mode for a 65 nm

Table 7 Performance of IntAcc-RCHC compared to RC4 and HC-128 in standalone mode

Cipher	Area (KGates)			Frequency (GHz)	Cycles per Byte	Throughput (Gbps)
	Seq.	Comb.	Total			
RC4 standalone	0.93	1.56	2.49	2.33	2	9.3
HC-128 standalone	1.95	7.47	9.42	1.67	1	13.36
IntAcc-RCHC	2.22	9.16	11.38	1.67	–	–
– RC4	–	–	–	–	2	6.68
– HC-128	–	–	–	–	1	13.36

target technology library. The area results are reported using equivalent 2-input NAND gates. The performance of IntAcc-RCHC is presented in Table 7, which shows the data for both standalone performance of the ciphers as well as within the combined framework.

Throughput comparison: In the current literature, the best existing design for RC4 hardware using block RAM is by Matthews [15]. This uses two 256-byte dual-port RAM blocks for the storage and offers a throughput of 1 byte per 3 cycles. In comparison, our design IntAcc-RCHC performs *1.5 times better* in terms of throughput—1 byte per 2 cycles. For HC-128, there is no RAM based hardware implementation in the literature. Thus, we compare our results with the software performance of the cipher, as calculated from the data reported [20] in the eSTREAM portfolio: 4.43 Gbps on Intel Pentium M, 5.96 Gbps on Intel Pentium 4, and 6.15 Gbps on AMD Athlon 64 X2 4200+ platform. In comparison, our proposed integrated design IntAcc-RCHC provides a net throughput of 13.36 Gbps, which is at least *2 times faster* than general purpose processors.

Area comparison: The sequential area of the combined design IntAcc-RCHC is less than the sum of RC4 and HC-128 individual designs as we could share the pipeline registers and the counter. The combinational blocks, e.g., the adder and logical operators could not be shared. In principle sharing the adder is possible at a loss of speed. On the other hand, the configuration decoding for the combined design is marginally more complex due to more number of configuration settings. As a result, combinational area increased slightly. The total core area still reports a reduction when compared to the sum of the standalone implementations.

Regarding the performance evaluation of IntAcc-RCHC, the overall area reduction should also take into account the complete sharing of SRAM by the two designs. Therefore, an accelerator design of HC-128, by employing a little area overhead (20.86 %) and without any runtime performance loss, is capable of accommodating RC4.

6 Conclusion

In this paper, we propose a novel idea for unified cryptographic hardware accelerator design based on the algorithmic and structural similarities between the ciphers to be implemented. As practical case studies of our proposal, we present HiPAcc-LTE, an integrated high performance hardware accelerator for 3GPP LTE stream ciphers SNOW 3G and ZUC, and IntAcc-RCHC, an integrated accelerator designed for

RC4 and HC-128, two of the most popular stream ciphers today. Through a careful design of the pipeline structure and storage organization, we achieve *significant advantages in terms of area* as well as *at least 1.5 to 2 times better throughput* compared to the state-of-the-art implementations, both in case of HiPacc-LTE and IntAcc-RCHC.

The design principle applied to the case studies performed in this paper can be exploited towards several similar hardware designs in the domain of cryptography. In particular, we would also like to explore the application of our approach towards an integrated accelerator for block ciphers and hash functions with structural similarities.

References

1. 3GPP TS 33.401 v11.0.1. 3rd Generation Partnership Project, Technical Specification Group Services and Systems Aspects. 3GPP System Architecture Evolution (SAE): Security Architecture. Release 11, June 2011
2. 3rd Generation Partnership Project: Long Term Evaluation Release 10 and beyond (LTE-Advanced). Proposed to ITU at 3GPP TSG RAN Meeting, Spain (2009)
3. Debraize, B., Corbella, I.M.: Fault analysis of the stream cipher Snow 3G. In: Fault Diagnosis and Tolerance in Cryptography (FDTC'09), September (2009)
4. Ekdahl, P., Johansson, T.: A new version of the stream cipher SNOW. In: Selected Areas in Cryptography (SAC'02), LNCS, vol. 2595, pp. 47–61. Springer, Heidelberg (2003)
5. Elliptic Technologies Inc. CLP-41: SNOW 3G flow through core. <http://www.elliptictech.com/en/products-a-solutions/hardware/cryptographic-engines/clp-41>. Accessed 5 Aug 2011
6. Elliptic Technologies Inc. CLP-400: SNOW 3G key stream generator. <http://www.elliptictech.com/en/products-a-solutions/hardware/cryptographic-engines/clp-400>. Accessed 5 Aug 2011
7. Elliptic Technologies Inc. CLP-403: SNOW 3G look aside core. <http://www.elliptictech.com/en/products-a-solutions/hardware/cryptographic-engines/clp-403>. Accessed 5 Aug 2011
8. Elliptic Technologies Inc. CLP-410: ZUC key stream generator. <http://www.elliptictech.com/en/products-a-solutions/hardware/cryptographic-engines/clp-410>. Accessed 5 Aug 2011
9. Elliptic Technologies Inc. CLP-411: ZUC look aside core. <http://www.elliptictech.com/en/products-a-solutions/hardware/cryptographic-engines/clp-411>. Accessed 5 Aug 2011
10. Elliptic Technologies Inc. CLP-412: ZUC flow through core. <http://www.elliptictech.com/en/products-a-solutions/hardware/cryptographic-engines/clp-412>. Accessed 5 Aug 2011
11. Intel Corporation: Intel advanced encryption standard instructions (AES-NI). <http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/>. Accessed 5 Aug 2011
12. IP Cores Inc: SNOW 3G encryption core. <http://ipcores.com/Snow3G.htm>. Accessed 5 Aug 2011
13. Kitsos, P., Selimis, G., Koufopavlou, O.: High performance ASIC implementation of the SNOW 3G stream cipher. In: IFIP/IEEE VLSI-SOC'08—International Conference on Very Large Scale Integration, Greece (2008)
14. Liu, Z., Zhang, L., Jing, J., Pan, W.: Efficient pipelined stream cipher ZUC algorithm in FPGA. In: First Int'l Workshop on ZUC Algorithm, China (2010)
15. Matthews, D.P., Jr.: System and method for a fast hardware implementation of RC4. US Patent Number 6549622, Campbell, CA, April. <http://www.freepatentonline.com/6549622.html> (2003). Accessed 5 Aug 2011
16. National Institute of Standards and Technology. Secure Hash Standard (SHS): Federal information processing standards publication (FIPS) 180-2. <http://csrc.nist.gov/publications/PubsFIPS.html>. Accessed 5 Aug 2011
17. Schaumont, P.R., Kuo, H., Verbauwhede, I.M.: Unlocking the design secrets of a 2.29 Gb/s Rijndael processor. In: Design Automation Conf. (DAC'02), USA (2002)
18. Schliebusch, O., Chattopadhyay, A., Steinert, M., Braun, G., Nohl, A., Leupers, R., Ascheid, G., Meyr, H.: RTL processor synthesis for architecture exploration and implementation. In: Design, Automation & Test in Europe (DATE'04)—Designers Forum, Paris, France (2004)

19. Sen, S., Gupta, Chattopadhyay, A., Khalid, A.: HiPAcc-LTE: an integrated high performance accelerator for 3GPP LTE stream ciphers. In: INDOCRYPT'11, LNCS, vol. 7107, pp. 196–215. Springer, Heidelberg (2011)
20. Software performance results from the eSTREAM Project. eSTREAM, the ECRYPT stream cipher project. <http://www.ecrypt.eu.org/stream/perf/#results>. Accessed 5 Aug 2011
21. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G specification. ETSI/SAGE Specification, Version: 1.1, 6 September 2006
22. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification. ETSI/SAGE Specification, Version: 1.5, 4 January 2011
23. Synopsys Processor Designer: Synopsys Inc. <http://www.synopsys.com/>. Accessed 5 Aug 2011
24. The current eSTREAM Portfolio. eSTREAM, the ECRYPT stream cipher project. <http://www.ecrypt.eu.org/stream/index.html>. Accessed 5 Aug 2011
25. Wu, H.: The stream cipher HC-128. The current portfolio of eSTREAM, the ECRYPT stream cipher project. <http://www.ecrypt.eu.org/stream/hcpf.html>. Accessed 5 Aug 2011