HiPAcc-LTE: An Integrated High Performance Accelerator for 3GPP LTE Stream Ciphers

Sourav Sen Gupta[1][*], Anupam Chattopadhyay[2], and Ayesha Khalid[2]

[1] Applied Statistics Unit, Indian Statistical Institute, Kolkata, India
[2] MPSoC Architectures, UMIC, RWTH Aachen University, Germany
`sg.sourav@gmail.com`, {`anupam, ayesha.khalid`}`@umic.rwth-aachen.de`

Abstract. Stream ciphers SNOW 3G and ZUC are the major players in the domain of next generation mobile security as both of them have been included in the security portfolio of 3GPP LTE-Advanced, the potential candidate for 4G mobile broadband communication standard. In this paper, 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. This is in comparison with the state-of-the-art implementations of the individual ciphers, both in the academic literature as well as in the commercial domain. We present detailed description of the design idea, optimization techniques and comparison results in this paper. 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: 3GPP, LTE-Advanced, Stream Cipher, SNOW 3G, ZUC, Hardware Implementation, ASIC, High Throughput, Area Efficiency.

1 Introduction

The world is expecting a completely new mobile broadband technology in the near future. The 3rd Generation Partnership Project (3GPP) had submitted Long Term Evaluation (LTE) Advanced [2] as a candidate for the 4G system, and LTE-Advanced (Release 10) [2] has been standardized by 3GPP as a major enhancement of their LTE standard. The main contenders for the new 3GPP LTE security suite had been EEA1/EIA1 [15] (identical to earlier protocols UEA2/UIA2) based on SNOW 3G [16], and EEA2/EIA2 based on block cipher AES-128. Later, a new set of security algorithms EEA3/EIA3 [17], based on another stream cipher ZUC [18], was proposed for inclusion in LTE, and it has been considered for verification and absorption in the LTE standards. At present, the block cipher AES-128 and the two stream ciphers SNOW 3G and ZUC constitute the core of the LTE security algorithms [1].

---

[*] This work was done during Sourav Sen Gupta's visit to MPSoC Architectures, UMIC, RWTH Aachen University, Germany as a Summer Intern during June–August 2011.

Motivation. It can be observed that the new security architecture for 4G is going to have two different stream ciphers, which are very unlikely to be used simultaneously. However, the hardware implementations of these two ciphers have always been done independently, as distinct cores. Be it in academic literature [12, 13] or in commercial products [5–11], even the single-platform implementations of LTE-Advanced security suite seem to use two distinct cores for the two stream ciphers. We, on the other hand, focus on merging the two towards a compact hardware solution for the LTE suite.

Our main objective, however, is to explore a more general problem in cryptographic hardware. One may want to implement two or more ciphers on the same platform, and in such a scenario, the following question arises:

> *If there is a requirement to implement an array of ciphers on the same platform, how should one approach the hardware design?*

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 first. Once this is accomplished, one may design an integrated custom accelerator for the ciphers such that each of the algorithms can be accessed individually. This approach offers the flexibility of

- sharing of resources, both storage and logic,
- optimization of mutual critical path,
- throughput vs. area optimization at the base level, and
- potential mechanism for 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. It is quite surprising why this kind of a hybrid approach has never been considered for integrated design of cryptographic accelerators.

Contribution. In this paper, we take up the LTE Stream Ciphers as a case study for our idea of hardware integration. There has been a few academic publications towards hardware implementation of the individual ciphers. Especially, Kitsos et al [12] provide us with a high performance ASIC implementation of SNOW 3G and recently Liu et al [13] 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. [11], 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 (HiPAcc-LTE) for SNOW 3G and ZUC (version 1.5, as in LTE Release 10 and beyond), 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 also 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 till date, we just leave the room for future fault protection requirements.

Long term vision of this hardware integration approach for cryptographic algorithms is to have 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 main theoretical idea of this paper. We discuss
- the approach for integration and restructure the hardware designs of the two ciphers to exploit the similarities to the fullest,
- construction of the optimal pipeline for the unified hardware, and
- the final top-level design and pipeline architecture for HiPAcc-LTE.

Section 4 deals with simulation, testing and synthesis of HiPAcc-LTE.
- We present a detailed account of the base implementation and further optimization of the proposed hardware.
- We also present a comparison of our design in multiple technologies with existing architectures in the academic and commercial domains.
- Furthermore, we provide a combined fault detection and correction facility in our architecture that prevents the known fault attack against SNOW 3G and allows prevention of future fault attacks against ZUC.

Section 5 concludes the paper by providing a future direction of research oriented around the idea of hardware integration proposed in this paper.

2 Preliminaries

Before diving into the main technical content of this paper, let us first recall the design and constructs of the stream ciphers SNOW 3G and ZUC. This individual overview of the ciphers will help us identify the similarities and dissimilarities in their designs, which will later lead to their high performance integration.

2.1 Brief Overview of SNOW 3G and ZUC

SNOW 3G [16] 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 illustrated in Fig. 1.
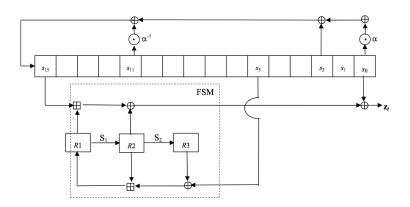

Fig. 1. SNOW 3G cipher in Keystream Generation mode [16].

ZUC [18] 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 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 1 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.
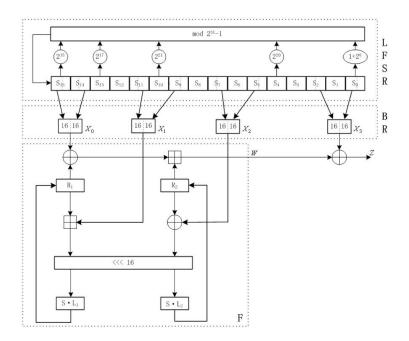

Fig. 2. ZUC cipher in Keystream Generation mode [18].

2.2 SNOW 3G and ZUC: Similarities and Dissimilarities in Design

In this section, we put the two ciphers SNOW 3G and ZUC side by side for a structural comparison in the designs. Observations made in this section will help us later in building the integrated platform for the LTE stream ciphers.

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.

A top level structure for both the ciphers can hence be represented as in Fig. 3. 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. 3, 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.
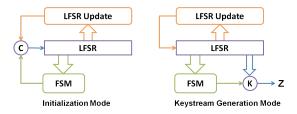

Fig. 3. Top level structure of both SNOW 3G and ZUC.

The key point to observe in Fig. 3 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 $R1 \to R2 \to R3 \to R1$ 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.

3 Integration of SNOW 3G and ZUC

In this section, we present our main idea behind the architectural integration of SNOW 3G and ZUC. 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.

3.1 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. 4, 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. 4.

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.
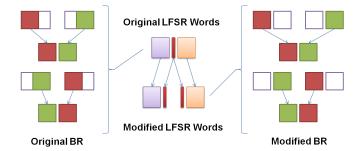
Fig. 4. Modified bit reorganization for ZUC after LFSR integration.

3.2 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 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 [16] uses S_R, S_Q and ZUC [18] 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 [16]. This saves a lot of computations after the memory read, and hence reduces the critical path to a considerable extent. We store the 8 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 4 times each during the FSM update. So, to parallelize the memory access, we store 4 copies of each table (thus 8 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 8 distinct memory locations, we have restricted the memory read requests to 1 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. 5.
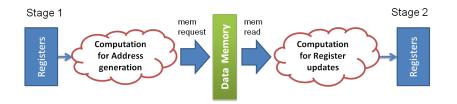

Fig. 5. Pipeline structure based on Memory Access.

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.

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.

3.3 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. 5).

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 Stage 2 to Stage 1 in our earlier pipeline organization (Fig. 5). 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 [16]. For an efficient implementation in hardware, we follow this idea and store the two tables MUL_{alpha} and DIV_{alpha} 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_{alpha} and DIV_{alpha}.
– 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 [13] had proposed a single adder implementation of this addition modulo prime, and this logic has also been included in the specifications [18]. 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 [13] for LFSR update of ZUC.

The initial idea for 5-stage pipeline is shown as Pipeline 1 in Fig. 6. All the adders are modulo prime, similar to the ones in [13], and the variables

a, b, c, d, e, f represent $s_0, 2^8 s_0, 2^{20} s_4, 2^{21} s_{10}, 2^{17} s_{13}, 2^{15} s_{15}$ (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.
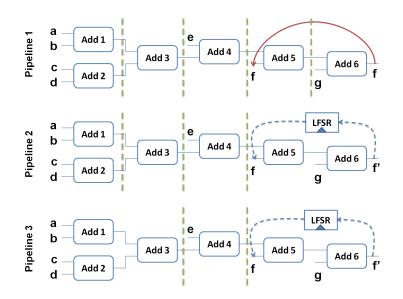

Fig. 6. Pipeline structure reorganization for LFSR update of ZUC.

However, Pipeline 1 creates a combinational loop between Stage 5 and Stage 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. 6, and it occurs due to mutual dependency of FSM and LFSR update during initialization. The authors of [13] 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. 6.

The reader may note that we have two adders (modulo prime p) in series at the last stage of Pipeline 2 (Fig. 6). 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. 6 as Pipeline 3. In the next section, we design the integrated pipeline structure combining all components.

3.4 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. 7.
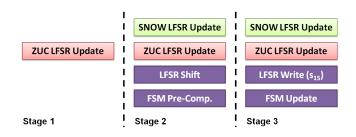

Fig. 7. Final 3-stage Pipeline Structure for the Integrated Design.

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 of these in Fig. 7. In the next section, we discuss the practical issues with the final ASIC implementation of our integrated hardware.

4 ASIC Implementation of the Integrated Hardware

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 [19]. 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. 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 [14], 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. Fig. 8 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. 8 to the original initialization routine of ZUC, as described in its specification [18].
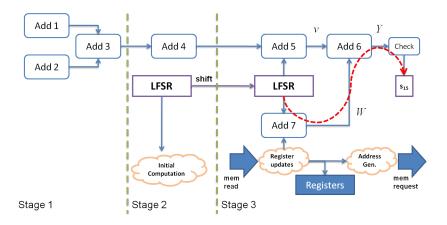

Fig. 8. Critical path in the Key Initialization of ZUC (curvy dashed line).

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 [18].

```
LFSR_Key_Initialization (W)
```
1. $v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + 2^8 s_0 + s_0 \pmod{2^{31} - 1}$
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. 8, the first five adders Add 1 to Add 5 are part of the general LFSR feedback loop in ZUC, and they compute the value

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

The LFSR is also accessed to run the FSM and the adder Add 7 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 6 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. 8. 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. 8, 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 Stage 4 and Stage 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 [13]. This looks like the circuit on the left of Fig. 9. 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. 9) 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 6 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.
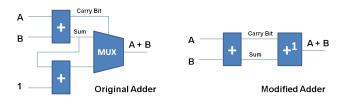
Fig. 9. Modulo p Adder optimization for ZUC.

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 6 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 our 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. 10.

Frequency	Area (equivalent NAND Gates)		
(MHz)	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

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

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.

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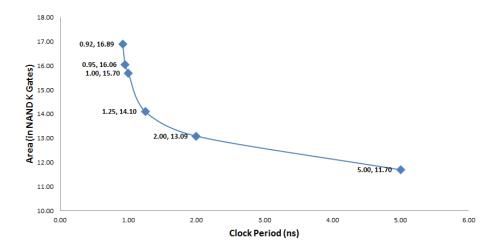
Fig. 10. Area-Time chart for HiPAcc-LTE (10 KByte memory) using 65 nm technology.

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 [16] 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 4 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_{alpha} and DIV_{alpha} 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.

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

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

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

It is expected that our design will be practically deployed in a system-on-chip setting, where the memory requirement of 3 KByte is quite reasonable. 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 stand-alone mode. The synthesis results in this direction are presented in Table 3.

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

Table 3. Synthesis results for stand-alone mode in HiPAcc-LTE with 3 KByte memory.

4.3   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 [13] 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 [12] 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 [12]: 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

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 stand-alone 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. [11] and Elliptic Tech Inc. [8] respectively. Both provide stand-alone 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 stand-alone 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.

Performance of Commercial Designs					
Cipher	Name of Design	Designer	Max. Freq. (MHz)	Throughput (Gbps)	Total Area (KGates)
SNOW 3G	SNOW3G1 [11]	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	Frequency (MHz)	Throughput (Gbps)	Memory (KGates)	Total Area (KGates)
SNOW 3G	3 KByte memory	1000	32.0	43.0	50.0
ZUC	3 KByte memory	1000	32.0	26.8	38.2
Both	3 KByte memory	1090	34.9	43.0	59.9
SNOW 3G	Hard macro	1650	52.8	-	18.1
ZUC	Hard macro	920	29.4	-	20.6
Both	Hard macro	920	29.4	-	23.9

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

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 [11].

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 stand-alone implementations of the ciphers.

For a very rough estimate, if one wants to achieve a comparable throughput (approx. 30 Gbps) using the existing stand-alone modules, then 4 parallel blocks of SNOW3G1 [11] and 2 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 23.9 KGates (almost 57% 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.

4.4 Fault Detection and Protection in HiPAcc-LTE

Till 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 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.

5 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 a practical case study of our proposal, we present HiPAcc-LTE, an integrated high performance hardware accelerator for 3GPP LTE stream ciphers SNOW 3G and ZUC. Through a careful design of the pipeline structure and storage organization, we achieve significantly higher throughput than state-of-the-art implementations.

A detailed physical synthesis and post-layout validation of the proposed design HiPAcc-LTE is on our roadmap. Furthermore, the design principle applied to the LTE stream ciphers can be exploited towards several similar hardware designs in the domain of cryptography. In particular, we would like to explore the application of our approach towards an integrated accelerator for other stream ciphers, 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. B. Debraize and I.M. Corbella. Fault Analysis of the Stream Cipher Snow 3G. Fault Diagnosis and Tolerance in Cryptography (FDTC '09), September, 2009.
4. P. Ekdahl and T. Johansson. A New Version of the Stream Cipher SNOW. Selected Areas in Cryptography (SAC'02), LNCS, Springer, Vol. 2595, pp. 47–61, 2003.
5. Elliptic Technologies Inc. CLP-41: SNOW 3G Flow Through Core. Available at `http://elliptictech.com/products-clp-41.php`. Retrieved on 5 August 2011.
6. Elliptic Technologies Inc. CLP-400: SNOW 3G Key Stream Generator.
7. Elliptic Technologies Inc. CLP-403: SNOW 3G Look Aside Core. Available at `http://elliptictech.com/products-clp-403.php`. Retrieved on 5 August 2011.
8. Elliptic Technologies Inc. CLP-410: ZUC Key Stream Generator. Available at `http://elliptictech.com/products-clp-410.php`. Retrieved on 5 August 2011.
9. Elliptic Technologies Inc. CLP-411: ZUC Look Aside Core. Available at `http://elliptictech.com/products-clp-411.php`. Retrieved on 5 August 2011.
10. Elliptic Technologies Inc. CLP-412: ZUC Flow Through Core. Available at `http://elliptictech.com/products-clp-412.php`. Retrieved on 5 August 2011.
11. IP Cores Inc. SNOW 3G Encryption Core. Available at `http://ipcores.com/Snow3G.htm`. Retrieved on 5 August 2011.
12. P. Kitsos, G. Selimis and O. Koufopavlou. High Performance ASIC Implementation of the SNOW 3G Stream Cipher. IFIP/IEEE VLSI-SOC '08 - International Conference on Very Large Scale Integration, Greece, 2008.
13. Z. Liu, L. Zhang, J. Jing and W. Pan. Efficient Pipelined Stream Cipher ZUC Algorithm in FPGA. First Int'l Workshop on ZUC Algorithm, China, 2010.
14. P.R. Schaumont, H. Kuo and I.M. Verbauwhede. Unlocking the design secrets of a 2.29 Gb/s Rijndael processor. Design Automation Conf. (DAC '02), USA, 2002.
15. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 1: UEA2 and UIA2 Specification. ETSI/SAGE Specification, Version: 1.1, September 6, 2006.
16. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification. ETSI/SAGE Specification, Version: 1.1, September 6, 2006.
17. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 1: 128-EEA3 and 128-EIA3 Specification. ETSI/SAGE Specification, Version: 1.5, January 4, 2011.
18. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification. ETSI/SAGE Specification, Version: 1.5, January 4, 2011.
19. Synopsys Processor Designer. Synopsys Inc. `http://www.synopsys.com/`.