An Efficient FPGA Architecture for Block Ciphering in Third Generation Cellular Networks

Tomás Balderas-Contreras and René A. Cumplido-Parra

Instituto Nacional de Astrofísica, Óptica y Electrónica
Luis Enrique Erro No. 1. Postal Code: 72840
Tonantzintla, Puebla. MEXICO
{balderas,rcumplido}@inaoep.mx

Abstract. Third generation cellular networks allow users to transmit information at high data rates, have access to IP-based networks deployed around the world, and implement sophisticated services. Not only is it necessary to develop new radio interface technologies and improve existing core networks to reach success, but guaranteeing confidentiality and integrity during transmission is a must. The KASUMI block cipher lies at the core of the integrity and confidentiality techniques designed for Universal Mobile Telecommunications System (UMTS) third generation networks. In turn, KASUMI implementations must reach high performance and have low power consumption in order to be adequate for network components. This paper describes a hardware architecture designed to perform the KASUMI algorithm efficiently, shows the convenience of taking advantage of the features present in FPGAs, and highlights a technique that might be used to design small reuse-based architectures implementing Feistel-like ciphering algorithms.

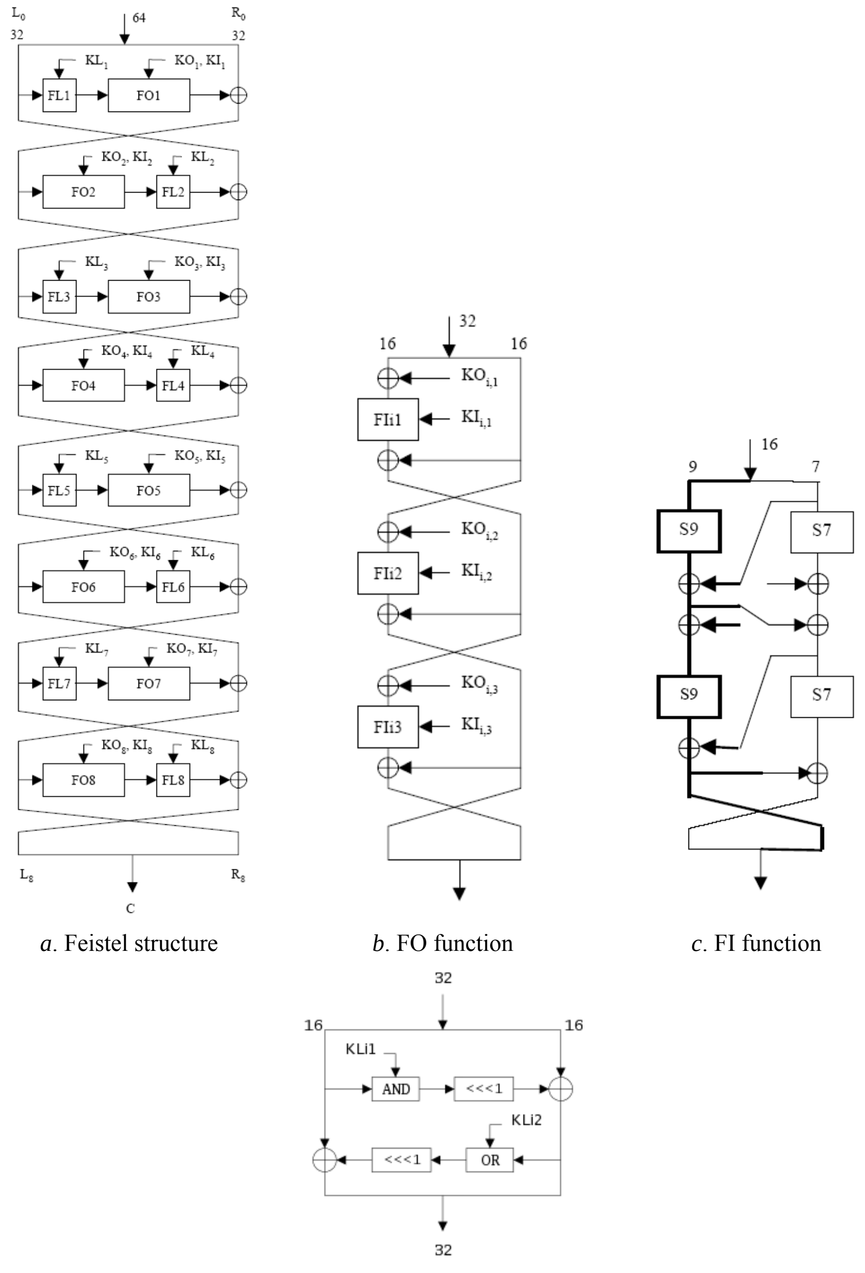
Resumen: Este artículo describe una implantación en hardware del algoritmo de cifrado de bloques KASUMI, el cual es la piedra angular de las operaciones de confidencialidad e integridad en redes celulares de tercera generación de tipo UMTS (Universal Mobile Telecommunications System). El diseño propuesto explota el principio de reutilización de componentes y las características presentes en FPGAs modernos. Además, la arquitectura resulta ser un buen balance entre alto desempeño y bajo consumo de recursos de hardware.

Keywords: 3G cellular networks, 3GPP, KASUMI, FPGA.

1 Introduction

The efforts towards the standardization of a block ciphering algorithm to be used in UMTS networks produced the specification of the KASUMI algorithm by the Third Generation Partnership Program (3GPP) [1]. KASUMI has a Feistel structure and operates on 64-bit data blocks under control of a 128-bit encryption key K [2]. KASUMI has the following features, as a consequence of its Feistel structure:

- Input plaintext is the input to the first round.
- Ciphertext is the last round's output.
- K is used to generate a set of round keys $\{KL_i, KO_i, KI_i\}$ for each round i.



a. Feistel structure *b*. FO function *c*. FI function

d. FL function

Fig. 1. The KASUMI block cipher

- Each round computes a different function, as long as the round keys are different.
- The same algorithm is used both for encryption and decryption.

KASUMI is based on a previous block cipher called MISTY1 [5]. MISTY1 was chosen as the foundation for the 3GPP ciphering algorithm due to its proven security against the most advanced methods to break block ciphers, namely cryptanalysis techniques. In addition, MISTY1 was heavily optimized for hardware implementation.

As can be seen in Fig. 1, the KASUMI block cipher has a different setup for the functions within rounds. For odd rounds the round-function is computed by applying the FL function followed by the FO function. For even rounds FO is applied before FL. FL, shown in Fig. 1*d*, is a 32-bit function made up of simple AND, OR, XOR and left rotation operations. FO, depicted in Fig. 1*b*, is also a 32-bit function having a three-round Feistel organization which contains one FI block per round. FI, see Fig. 1*c*, is a non-linear 16-bit function having itself a four-round Feistel structure; it is made up of two nine-bit substitution boxes (S-boxes) and two seven-bit S-boxes. Fig. 1*c* shows that data in the FI function flow along two different paths: a nine-bit long path (thick lines) and a seven-bit path (thin lines). Notice that in Feistel structures, such as the ones used in this algorithm, each round's output is twisted before being applied as input to the following round. After completing eight rounds KASUMI produces a 64-bit long ciphertext block corresponding to the input plaintext block.

Several proposals have been published previously that use different approaches to implement KASUMI in hardware, ranging from reuse techniques, addition of internal registers to reduce critical path and pipelined designs. After analyzing these proposals one can confirm the fact that there is an important and unavoidable tradeoff between performance and complexity in terms of area. The goals of this paper are twofold. First, to reach a good balance between high performance and low complexity during the implementation of KASUMI by taking advantage of the resources present in modern FPGA and the software tools used to implement designs for these devices. Second, to present the methodology used to design a compact datapath that reuses its components again and again to carry out the encryption process. Similar techniques can be applied to other Feistel-like algorithms.

The rest of this document is organized as follows: section 2 describes the design principles of the proposed architecture; section 3 concerns the implementation phase and describes the platform used for this project, provides results from the synthesis process and compares this information with the information provided by other works; finally, section 4 concludes.

2 Design principles

2.1 Datapath for the FO function

The main principle to design for reuse is to specify an architecture consisting of some or all of the components that are needed to perform a round. This design is used every clock cycle in such a way that the output at the end of one cycle is the input for the next cycle. The fewer the components, the larger the number of cycles needed to carry out the whole ciphering process for one block. Also, the fewer cycles the architecture requires to perform the process, the more complex the architecture is in terms of area.

The approach used in this paper is considered to be a good tradeoff between number of cycles and area complexity in the platform of choice. Instead of simplifying the algorithm at a lower level, at the FI level as in [6], the manipulation is carried out at a higher level, at the FO-function level. Fig. 2 depicts the process followed to design a datapath that reuses components within FO function. Fig. 2*a* illustrates an alternative parallel view of the FO function depicted in Fig. 1.

- In Fig. 2*b* two XOR gates are added to FO in order to make upper and lower sections more similar without modifying the behavior of the function. If these two parts were structurally the same, it would be possible to reduce the architecture to only one section, which carries out the whole FO function after two cycles.
- Lower section in Fig. 2*b* needs a right FI function block in order to be structurally identical to upper section in this modified FO block. In Fig. 2*c* an additional FI block is added to the lower section. The multiplexors in each section allow selecting the appropriate data flow.
- The whole datapath in Fig. 2*c* is now ready to be simplified. Fig. 2*d* shows the final design, which takes two cycles to complete the FO function. Notice that multiplexors are necessary in order to supply the correct values both to XOR gates and FI module depending on if the datapath is in the first or second cycle.

Also notice that the datapath depicted in Fig. 2*d* contains only one FI block, called dpFI, instead of two as in the previous diagrams. This situation is explained in more detail next because it constitutes another special feature of the design proposed in this document.

The control for this module is implemented as a finite state machine that sets the multiplexors' selector input properly each cycle. Since the design takes two cycles to complete its processing, it requires a two-state control.

2.2 Datapath for the FI function

Fig. 2*c* shows that the FO module requires two FI blocks to work properly. Since FI contains two seven-bit S-boxes and two nine-bit S-boxes, the simplified datapath that takes two cycles to complete the FO function requires a total number of eight S-boxes. Implementing S-boxes using four 128 × 7 ROMs and four 512 × 9 ROMs is such an expensive choice in terms of area to be considered as viable. The solution proposed is to map the S-boxes to dual-port ROMs, which decreases by two the number of ROMs required. The use of this technique exploits the principle of reuse even further because the same S-box is now able to meet two requests at the same time.

Consider two instances of the FI block depicted in Fig. 1*c*; next replace each pair of S9 S-boxes located in the same position in both FI blocks by a single dual-port S-box, and repeat this procedure with the rest of the pairs of S9 S-boxes and the pairs of S7 S-boxes. The result is the datapath illustrated in Fig. 3, which only contains two dual-port S9 S-boxes and two dual-port S7 S-boxes, and combines two FI functions into one. As before, the thick lines highlight the flow of nine-bit long signals and the thin lines highlight the flow of seven-bit long signals.

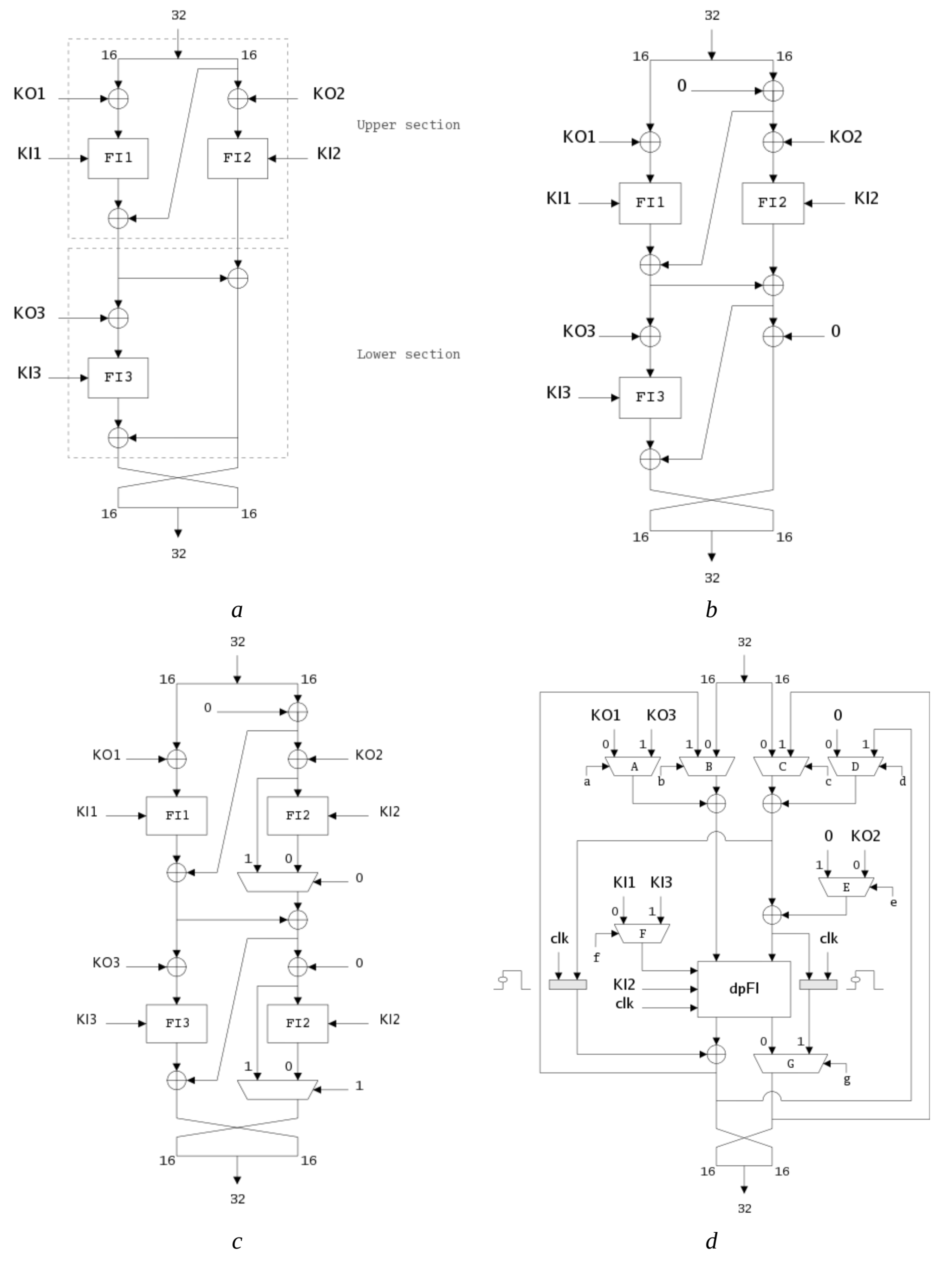


Fig. 2. Sequence of steps to design the datapath that reuses components in FO function

There are several notes to point out concerning this design. First, the four dual-port ROMs used to implement the S-boxes are intended to be mapped to embedded memory blocks inside the FPGA during the implementation phase. Second, the common situation is that these embedded memory blocks are synchronous, and since this dual-port FI datapath is required to provide its results within the range of one clock cycle, the upper S-boxes are designed to be negative edge-triggered, whereas the lower S-boxes are designed to be positive edge-triggered, as indicated in Fig. 3. Third, there are plenty of registers throughout the design depicted in Fig. 3, they are colored in grey and their purpose is to synchronize input data with the values provided by the upper and lower S-boxes. This situation is very similar to that present in pipelined datapaths.

Notice that the datapath in Fig. 2*d* also has positive edge-triggered registers used to synchronize input data used by logic that is located further away the dual-port FI module in the datapath. Indeed, every input signal that is to be used after a component containing the dual-port FI module must be synchronized with the data provided by the dual-port FI module by means of registers, either positive edge-triggered or negative edge-triggered.

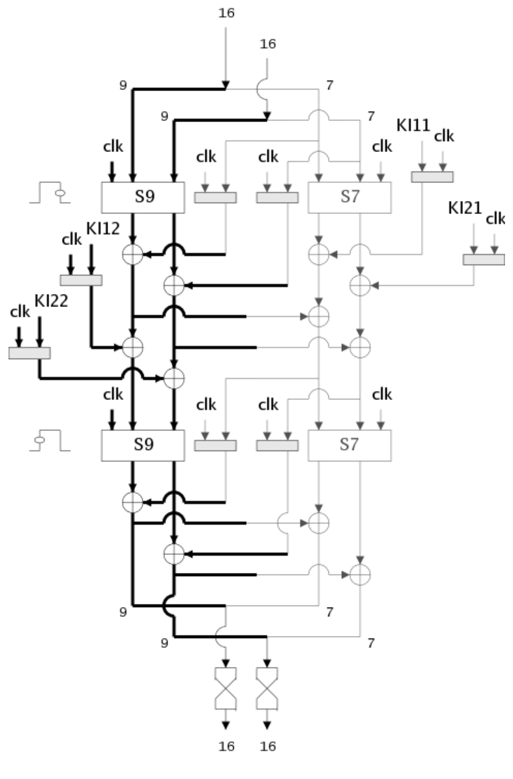


Fig. 3. Datapath implementing two merged FI function blocks

2.3 Datapath for the round logic

The round logic, depicted in Fig. 4, is the highest-level component of the KASUMI design proposed in this paper. During the first two cycles it receives input data from outside by selecting the zero input in multiplexors A and B, and performs an odd-round operation by selecting zero input both in multiplexor C and in multiplexor D. During the next 14 cycles the outputs yielded by the datapath each cycle are fed back to its inputs. The same data must be present at the datapath's inputs during two cycles in order to carry out the correct processing; that is why there is a third registered input in both input multiplexors.

Notice that input data used after the FO function module, which in turn contains the dpFI function module, are synchronized using registers as shown in Fig. 4. Registers are colored in grey in this figure as well.

The control for this module is implemented as a finite state machine that sets each multiplexor's selector input properly each cycle. The datapath in Fig. 4 requires 16 cycles to fulfill the encryption process for every plaintext block. Therefore, the finite state control is made up of 16 states and controls the four selectors.

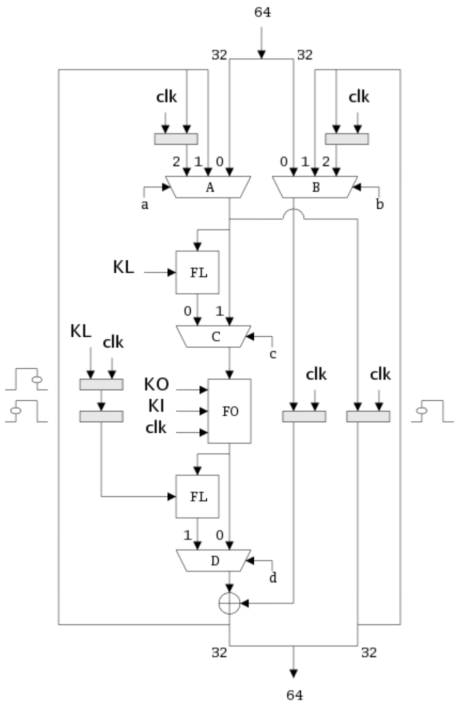


Fig. 4. Datapath for the round logic

2.4 The key scheduler

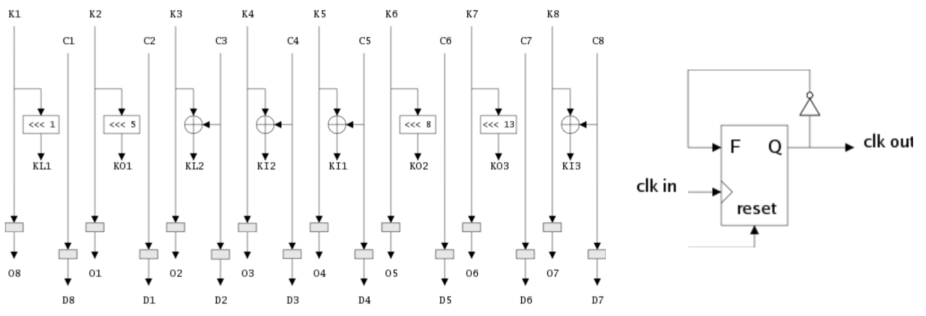
The key scheduler receives the 128-bit initial input key K and generates the round keys KL (32-bit long), KO (48-bit long) and KI (48-bit long) for each of the eight rounds. All of the previous figures depict how these round keys are used within each of the function modules. Each round key is split into two or three 16-bit parts, and these parts are the ones directly computed by the key scheduler. The input K key is split into eight 16-bit parts K_i, $1 \leq i \leq 8$, and then the scheduler performs left rotation operations ("<<<") and computes the K_i' values, which are defined as follows:

$$K_i' = K_i \oplus C_i \qquad 1 \leq i \leq 8; \tag{1}$$

where C_i is a constant specified in [2].

Fig. 5*a* illustrates the key scheduler component developed for this project, which adapts easily to different implementation schemes. For this design, the outputs are fed back to the inputs. The inputs are the initial key as an array of 16-bit values and the array of 16-bit constants. In addition to yielding the round keys in a combinational way, this component outputs its input arrays rotated to the left one position.

Notice that the design for the round logic described previously requires that each set of round keys is available during two cycles. Adding logic to the key scheduler to keep its output round keys without change for two cycles might result in a complex and expensive circuit. The most efficient way to fulfill the requirement is to connect the key scheduler's clock input to the output of the divide-by-2 clock divider in Fig. 5*b*. This technique preserves the key scheduler's simplicity without affecting the ciphering process.



a. Key scheduler design *b*. Divide-by-2 clock divider

Fig. 5. The components of the key scheduling system

3 Implementation

The architecture is designed in VHDL and synthesized using the Xilinx Synthesis Technology (XST) software synthesis tools. The device of choice is the XCV300E-8-

BG432 belonging to the Virtex-E family of devices [7]. The main reason for choosing this family is that the designs reported in the related papers are implemented on Virtex-E devices, so in order to make a fair comparison the design reported here is implemented on a Virtex-E device as well.

Virtex-E devices contain large blocks of embedded SelectRAM memories. Each of these blocks is a fully synchronous dual-port (True Dual Port) 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured in an independent fashion. The specific device considered in this project has 32 blocks of embedded SelectRAM, which provides a total number of 131072 bits of embedded memory.

3.1 Synthesis and results

Table 1 shows the results of the synthesis process concerning utilization of FPGA resources. It can be noticed that the percentage of resources used by the design in each category does not surpass 20%. By analyzing this information it is possible to conclude that the design is quite compact and fits well in any device without occupying a great number of resources.

Table 1. Synthesis results concerning area complexity

Category	Number of elements used	Total number of elements available	Percentage of use
Slices	488	3072	15%
Slice Flip Flops	566	6144	9%
4-input LUTs	898	6144	14%
BRAMs	6	32	18%
GCLKs	1	4	25%

Notice the number of SelectRAM blocks used by the design. Subsection 2.2 states that only four S-boxes are required in this design. Intuitively, these S-boxes would be mapped to four internal embedded memory blocks (SelectRAM), but table 1 shows that the design actually requires 6 of them. Consider the number of bits needed to implement each S-box: $128 \times 7 = 896$ bits for S7 S-boxes and $512 \times 9 = 4608$ bits for S9 S-boxes. A S7 S-box fits well in a 4096-bit SelectRAM block whereas a 4608-bit S9 S-box is far larger than a 4096-bit SelectRAM block, so two of them are required to implement one S9 dual-port S-box ROM.

Table 2 compares the results for this work with the results for other proposals published previously which implement a reuse approach. The work in [6] describes two implementations of KASUMI taking 56 and 32 cycles to cipher one block, results for a third implementation that takes 8 cycles are provided as well. Notice that the design that takes the largest number of cycles to process a block (56 cycles) is the one that requires the least number of slices and the one with the highest frequency. However, its throughput is poor, one of the lowest, as a consequence of the number of cycles needed to perform the processing of one block. The work reported in [4] takes 40 cycles to cipher one block, is the second most expensive in terms of area, due to it reuses two complete rounds, its throughput is comparable to the throughput of the slowest design in [6] and is reported to require 24 SelectRAM blocks to store the S-boxes.

The work in [3] is a combination of two approaches: pipeline and reuse, it is the most expensive design in terms of area but one of the fastest implementations. It is able to process four plaintext blocks at the same time, which increases performance dramatically.

Table 2. Comparison with other implementations

Proposal	Latency	Area (slices)	Frequency (MHz)	Throughput (Mbps)	HW efficiency (kbps/slice)	Number of S-boxes	Number of block RAMs
Work in [6]	56	368	68.13	77.86	211.58	2	N/A
	32	370	58.06	116.12	313.84	4	N/A
	8	588	33.14	265.12	450.88	12	N/A
Work in [4]	40	749	35.35	70.70	94.39	24	24
Work in [3]	pipeline	1100	33	234	212.73	12	N/A
This work	16	488	41.14	164.54	337.17	4	6

N/A = Not applicable

The design proposed in this paper features a novel strategy to deal with the reuse principle, is far cheaper than the two fastest options in terms of area, has one of the highest clock frequencies, and one of the highest throughputs. The main contribution of the design just described is that it achieves such a high performance with fewer hardware resources. The keys to achieve this good balance between speed and area complexity are the sharing of a high-level component (FO in this case), the use of dual-port SelectRAM blocks and a simplified design of the key scheduler.

4 Conclusions

This document presents a novel hardware implementation of the KASUMI block cipher designed using the principle of reuse of components. The architecture developed turns out to be a good balance between high performance and low complexity in area as a result of taking advantage of certain features present in modern FPGAs and some designs strategies. The main features of the architecture proposed are: reuse of higher-level components of the block cipher, which reduces the number of total cycles needed to carry out the process, mapping of the S-boxes to embedded dual-port memory blocks, and the design of a simple key scheduler that takes advantage of a clock-division technique. A similar technique to the one used in this work to design a shared datapath might be used to implement other Feistel-like block cipher algorithms. The design presented here can be implemented in other platforms such as Application Specific Integrated Circuit (ASIC) by carrying out the corresponding implementation process. The design proposed in this document might be incorporated in UMTS network components such as mobile equipment or Radio Network Controllers (RNCs), whether as a dedicated coprocessor or as a functional unit within a larger processor.

5 Acknowledgments

This work was sponsored by the scholarship number 171498 granted by CONACyT.

References

1. 3rd Generation Partnership Program. 3GPP Home Page. http://www.3gpp.org
2. 3rd Generation Partnership Program. Document 2: KASUMI Specification. Technical Specification 35.202. Release 5.Version 5.0.0.
3. Kim, H., et al.: Hardware Implementation of the 3GPP KASUMI Crypto Algorithm. Proc. of the 2002 International Technical Conference on Circuits/Systems, Computers and Communications ITC-CSCC-2002 (2002) 317-320.
4. Marinis, K., et al.: On the Hardware Implementation of the 3GPP Confidentiality and Integrity Algorithms. Proc. of the 4th International Conference on Information Security ISC 2001. LNCS 2200/2001. Springer-Verlag (2001) 248-265.
5. Matsui, M.: New Block Encryption Algorithm MISTY. Proc. of the 4th International Fast Software Encryption Workshop FSE'97. LNCS 1267/1997. Springer-Verlag (1997) 54-68.
6. Satoh, A., Morioka, S.: Small and High-Speed Hardware Architectures for the 3GPP Standard Cipher KASUMI. Proc. of the 5th International Conference on Information Security ISC 2002. LNCS 2433/2002. Springer-Verlag (2002) 48-62.
7. Xilinx, Inc.: Virtex-E 1.8 V Field Programmable Gate Arrays. v2.6. Product Specification (2002).