Efficient FPGA Implementations of Block Ciphers KHAZAD and MISTY1

by François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater and Jean-Didier Legat *

Abstract. The technical analysis used in determining which of the NESSIE candidates will be selected as a standard block cipher includes efficiency testing of both hardware and software implementations of candidate algorithms. Reprogrammable devices such as Field Programmable Gate Arrays (FPGA's) are highly attractive options for hardware implementations of encryption algorithms and this report investigates the significance of FPGA implementations of the block ciphers KHAZAD and MISTY1. A strong focus is placed on high throughput circuits and we propose designs that unroll the cipher rounds and pipeline them in order to optimize the frequency and throughput results. In addition, we implemented solutions that allow to change the plaintext and the key on a cycle-by-cycle basis with no dead cycle. The resulting designs fit on a VIRTEX1000 FPGA and have throughput between 8 and 9 Gbits/s. This is an impressive result compared with existing FPGA implementations of block ciphers within similar devices.

1 Introduction

The NESSIE project is about to put forward a portfolio of strong cryptographic primitives that has been obtained after an open call and been evaluated using a transparent and open process. These primitives include block ciphers, stream ciphers, hash functions, MAC algorithms, digital signature schemes, and public-key encryption schemes. The technical analysis used in determining which of the NESSIE candidates will be selected as a standard block cipher includes efficiency testing of both hardware and software implementations of candidate algorithms.

NESSIE candidate KHAZAD is a 64-bit block cipher that accepts a 128-bit key. Although KHAZAD is not a Feistel cipher, its structure is designed so that by choosing all round transformations components to be involutions, the inverse operation of the cipher differs from the forward operation in the key scheduling part only. This property makes it possible to reduce the required chip area in hardware implementations. The overall cipher design follows the Wide Trail Strategy, favours component reuse, and permits a wide variety of implementation

* UCL Crypto Group
Laboratoire de Microelectronique
Universite Catholique de Louvain
Place du Levant, 3, B-1348 Louvain-La-Neuve, Belgium
`{standaert,rouvroy,quisquater,legat}@dice.ucl.ac.be`

tradeoffs. Encryption algorithm MISTY1 is a 64-bit block cipher with a 128-bit key and a variable number of rounds. Mitsuru Matsui, the designer (1996), recommends a 8-round version. It is a Feistel cipher that allows very efficient hardware implementations. MISTY1 is designed on the basis of the theory of provable security against differential and linear cryptanalysis. In this report, we study the suitability of KHAZAD and MISTY1 for hardware implementations. Fast encryption modules are detailed and compared to AES RIJNDAEL and SERPENT in an effort to determine the efficiency of NESSIE candidates for hardware implementations within commercially available FPGA's.

This report is organized as follows. The description of the hardware, synthesis tools and implementation tools is in section 2. Section 3 gives a short mathematical description of KHAZAD and we propose a description of the diffusion layer that allows efficient pipelining. Our implementations of KHAZAD are in section 4. Section 5 gives a short mathematical description of MISTY1 and the corresponding implementations are in section 6. Comparisons with RIJNDAEL and SERPENT appear in section 7. Finally, conclusions are in section 8.

2 Hardware description

All our implementations were carried out on a XILINX VIRTEX1000BG560-6 FPGA. In this section, we briefly describe the structure of a VIRTEX FPGA as well as the synthesis and implementation tools that were used to obtain our results.

Configurable Logic Blocks (CLB's). The basic building block of the VIRTEX CLB is the logic cell (LC). A LC includes a 4-input function generator, carry logic and a storage element. The output from the function generator in each LC drives both the CLB output and the D input of the flip-flop. Each VIRTEX CLB contains four LC's, organized in two similar slices. Figure 1, shows a detailed view of a single slice. Virtex function generator are implemented as 4-input look-up tables (LUT's). In addition to operate as a function generator, each LUT can provide a 16×1-bit synchronous RAM. Furthermore, the two LUT's within a slice can be combined to create a 16×2-bit or 32×1-bit synchronous RAM or a 16×1-bit dual port synchronous RAM. The VIRTEX LUT can also provide a 16-bit shift register.

The storage elements in the VIRTEX slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by the function generators within the slice or directly from slice inputs, bypassing function generators.

The F5 multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine bits. Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions up to 19 bits.

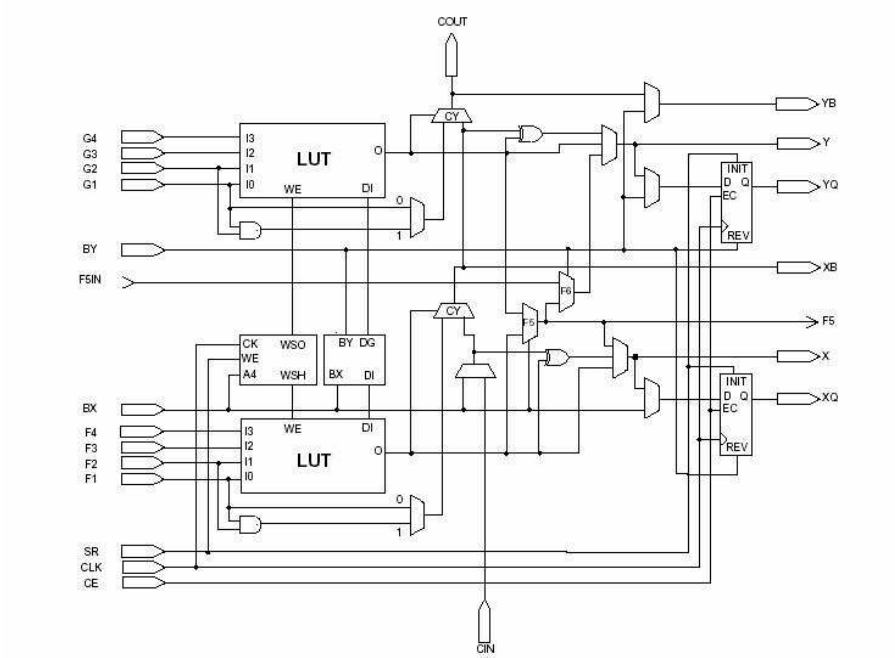

Fig. 1. The VIRTEX slice.

The arithmetic logic also includes a XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementations.

Finally, VIRTEX FPGA's incorporate several large RAM blocks. These complement the distributed LUT implementations of RAM's. Every block is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently.

Target FPGA. A VIRTEX1000BG560-6 FPGA contains 12288 slices and 32 RAM blocks, which means 24576 LUT's and 24576 flip-flops. In the following report, we compare the number of LUT's, registers and slices. We also evaluate the delays and frequencies thanks to our synthesis and implementation tools. The synthesis was performed with FPGA Express (SYNOPSYS) and the implementation with XILINX ISE-4. Finally, our circuits models were described using VHDL.

3 Block cipher description: KHAZAD

KHAZAD is an iterated block cipher that operates on a 64-bit cipher state represented as vectors in $GF(2^8)^8$. It uses a 128-bit key represented as a vector in $GF(2^8)^{16}$, and consists of a serie of applications of a key-dependent round transformation to the cipher state. In the following, we will individually define the component mappings and constants that build up KHAZAD, then speciphy the complete cipher in terms of these components.

Notation. Let s be a cipher state or a key $\in GF(2^8)^8$, then s_i is the i-th byte of the state s and $s_i(j)$ is the j-th bit of this byte.

The nonlinear layer γ. Function $\gamma : GF(2^8)^8 \rightarrow GF(2^8)^8$ consist of the parallel application of a non-linear substitution box S:

$$\gamma(a) = b \Leftrightarrow b_i = S[a_i], 0 \leq i \leq 7 \tag{1}$$

The substitution box is illustrated on figure 2, where P and Q are 4-bit input $\times$ 4-bit output look up tables. They were defined in order to optimize the resistance against differential and linear cryptanalysis and allow efficient hardware implementations.

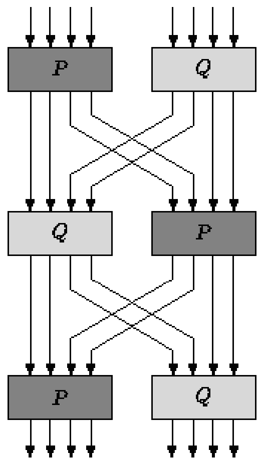

Fig. 2. The KHAZAD substitution box.

The diffusion layer θ. Function $\theta : GF(2^8)^8 \rightarrow GF(2^8)^8$ is a linear mapping based on a $[16, 8, 9]$ MDS[1] code:

$$\theta(a) = b \Leftrightarrow b = a.H \tag{2}$$

With:

$$H = \begin{bmatrix} 01 & 03 & 04 & 05 & 06 & 08 & 0B & 07 \\ 03 & 01 & 05 & 04 & 08 & 06 & 07 & 0B \\ 04 & 05 & 01 & 03 & 0B & 07 & 06 & 08 \\ 05 & 04 & 03 & 01 & 07 & 0B & 08 & 06 \\ 06 & 08 & 0B & 07 & 01 & 03 & 04 & 05 \\ 08 & 06 & 07 & 0B & 03 & 01 & 05 & 04 \\ 0B & 07 & 06 & 08 & 04 & 05 & 01 & 03 \\ 07 & 0B & 08 & 06 & 05 & 04 & 03 & 01 \end{bmatrix}$$

[1] MDS: Maximum Distance Separable.

We propose the following description of the diffusion layer that allows to introduce pipeline levels inside the layer:

$$
\begin{aligned}
b_0 &= a_0 \oplus a_1 \oplus a_3 \oplus a_6 \oplus a_7 \oplus X(a_1 \oplus a_4 \oplus a_6 \oplus a_7) \oplus X^2(a_2 \oplus a_3 \oplus a_4 \oplus a_7) \oplus X^3(a_5 \oplus a_6) \\
b_1 &= a_0 \oplus a_1 \oplus a_2 \oplus a_6 \oplus a_7 \oplus X(a_0 \oplus a_5 \oplus a_6 \oplus a_7) \oplus X^2(a_2 \oplus a_3 \oplus a_5 \oplus a_6) \oplus X^3(a_4 \oplus a_7) \\
b_2 &= a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus X(a_3 \oplus a_4 \oplus a_5 \oplus a_6) \oplus X^2(a_0 \oplus a_1 \oplus a_5 \oplus a_6) \oplus X^3(a_4 \oplus a_7) \\
b_3 &= a_0 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus X(a_2 \oplus a_4 \oplus a_5 \oplus a_7) \oplus X^2(a_0 \oplus a_1 \oplus a_4 \oplus a_7) \oplus X^3(a_5 \oplus a_6) \\
b_4 &= a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_7 \oplus X(a_0 \oplus a_2 \oplus a_3 \oplus a_5) \oplus X^2(a_0 \oplus a_3 \oplus a_6 \oplus a_7) \oplus X^3(a_1 \oplus a_2) \\
b_5 &= a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus X(a_1 \oplus a_2 \oplus a_3 \oplus a_4) \oplus X^2(a_1 \oplus a_2 \oplus a_6 \oplus a_7) \oplus X^3(a_0 \oplus a_3) \\
b_6 &= a_0 \oplus a_1 \oplus a_5 \oplus a_6 \oplus a_7 \oplus X(a_0 \oplus a_1 \oplus a_2 \oplus a_7) \oplus X^2(a_1 \oplus a_2 \oplus a_4 \oplus a_5) \oplus X^3(a_0 \oplus a_3) \\
b_7 &= a_0 \oplus a_1 \oplus a_4 \oplus a_6 \oplus a_7 \oplus X(a_0 \oplus a_1 \oplus a_3 \oplus a_6) \oplus X^2(a_0 \oplus a_3 \oplus a_4 \oplus a_5) \oplus X^3(a_1 \oplus a_2)
\end{aligned}
$$

Where $b_7, b_6, ..., b_0$ represent the eight bytes of the cipher state and X is defined at the byte level as: $X : GF(2^8) \rightarrow GF(2^8) : X(a) = b \Leftrightarrow$

$$
\begin{aligned}
b(7) &= a(6) \\
b(6) &= a(5) \\
b(5) &= a(4) \\
b(4) &= a(3) \oplus a(7) \\
b(3) &= a(2) \oplus a(7) \\
b(2) &= a(1) \oplus a(7) \\
b(1) &= a(0) \\
b(0) &= 0 \oplus a(7)
\end{aligned}
$$

Finally, we define functions $X^2 \equiv X \circ X$ and $X^3 \equiv X \circ X \circ X$.

The key addition $\boldsymbol{\sigma}$. The affine key addition $\sigma[k] : GF(2^8)^8 \rightarrow GF(2^8)^8$ consists of the bitwise addition (exor) of a key vector $k \in GF(2^8)^8$:

$$\sigma[k](a) = b \Leftrightarrow b_i = a_i \oplus k_i, 0 \leq i \leq 7 \tag{3}$$

The round constants. The constant for the r-th round is a vector $c^r \in GF(2^8)^8$, defined as:

$$c_i^r = S[8r + i], 0 \leq r \leq 8, 0 \leq i \leq 7 \tag{4}$$

The round function $\boldsymbol{\rho}$. The r-th round function is the composite mapping $\rho[k] : GF(2^8)^8 \rightarrow GF(2^8)^8$, parameterized by the key vector $k \in GF(2^8)^8$ and given by:

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \gamma \tag{5}$$

The key schedule. The key schedule expands the cipher key $K \in GF(2^8)^{16}$ into a sequence of round keys $K^0, K^1, ...; K^8$, plus two initial values K^{-2} and K^{-1} corresponding to the most and least significant parts of the cipher key K. Every round key is an element of $GF(2^8)^8$ that we derive as follows:

$$K^r = \rho[c^r](K^{r-1}) \oplus K^{r-2}, 0 \leq r \leq 8 \tag{6}$$

The complete cipher. KHAZAD is defined for the cipher key K as the transformation $\text{KHAZAD}[K] = \alpha[K^0, K^1, ..., K^8]$ applied to the plaintext, where

$$\alpha[K^0, K^1, ..., K^8] = \sigma[K^8] \circ \gamma \circ (\bigcirc_{r=1}^{7} \rho[K^r]) \circ \sigma[K^0] \tag{7}$$

Our implementation is based on this description of KHAZAD.

4 Implementation: KHAZAD

4.1 Objectives

FPGA's are very efficient devices and they are suitable for high work frequencies. As opposed to custom hardware or software implementations, little work exist in the area of block cipher implementations within existing FPGA's. Results available in the public literature sometimes mention encryption rates comparable with software ones. We believe that these performances can be greatly improved using today's technology as soon as inherent constraints of FPGA's are taken into account.
The VIRTEX slice offers great flexibility to implement various logic functions, but it also constraints the designer to an efficient usage of its resources. Regarding the inner structure of KHAZAD, we determined that an optimal circuit should limit its critical path inside one slice, without consuming slices for register usage only.

4.2 Components

Table 1 evaluates the hardware cost of some basic elements of KHAZAD. Their structure, very close to the VIRTEX LUT, allow a direct and efficient implementation. Practically, we optimized our circuit by keeping its critical path inside the slice of Fig 1, and making an efficient use of its registers.

Table 1. Some combinatorial components of KHAZAD.

Component	Nbr of LUT
X	3
S	24
γ layer	192
Key addition σ	64

Actually, the most critical function in terms of implementation is the diffusion layer θ. In the precedent section, we gave a combinatorial description of it that allow us to consider different pipeline levels. For efficiency purposes, we also combined θ with the key addition layer σ, because of their relevant compatibility. Figure 3 illustrates the computation of an output byte b_0 of the diffusion layer θ combined with key addition σ. The key point of this architecture is the central

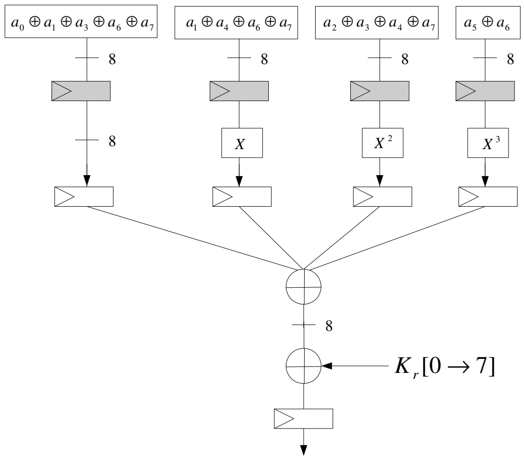

Fig. 3. KHAZAD: output byte b_0 of the diffusion layer θ combined with key addition σ.

bitwise XOR operation between 4 bytes. As the VIRTEX slice contains, a 4-input LUT and an additional XOR gate, we can efficiently combine this operation with the bitwise key addition and perform the resulting task in one cycle.

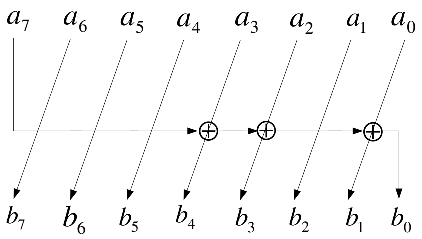

Fig. 4. The function X of KHAZAD.

The upper part of θ don't permit this kind of optimization. Looking at Figures 3, 4, we see that the byte $a_2 \oplus a_3 \oplus a_4 \oplus a_7$ is an input of function $X^2 = X \circ X$. This can't be done inside one slice. Consequently, we considered two circuits depending on the diffusion-addition layer implemented:

1. A fast and expensive implementation using three registers levels inside the layer.
2. A slower but less expensive implementation where the grey register of θ is removed.

A tradeoff has to be done between a low-delay (pipeline of X, X^2, X^3 functions) and a low area where we avoid the implementation of useless registers in the left branch of θ.

Table 2. KHAZAD: implementations of the diffusion layer θ combined with key addition σ.

Type	Nbr of LUT	Nbr of registers	Estimated delay (ns)
Fast implementation	384	576	5.2
Low area implementation	384	320	7.3

Table 2 gives the implementation results of our two implementations. In this section, the delay is estimated after synthesis[2]. Note that in the fast implementation, the critical path corresponds to a look up table and an exor operation in the left branch of θ.

4.3 The round and key round functions

Based on the above components, we propose two solutions for the round and key round functions, with a difference of one register level. Figure 5 illustrates the round function of KHAZAD. Figure 6 illustrates its key round. Depending on the use of the grey register, we obtain the results of table 3.

Table 3. KHAZAD: implementations of the round and key round.

Type	Nbr of LUT	Nbr of registers	Estimated delay (ns)
Fast round	576	768	5.5
Low area round	576	512	7.3
Fast key round	768	832	5.6
Low area key round	756	576	7.3

4.4 The complete cipher

The implementation of the complete KHAZAD cipher directly results from the precedent descriptions. Our results are summarized by the next figure and table: Figure 7 illustrates our two versions of the complete cipher. Finally, table 4 summarizes our implementation results for the block cipher KHAZAD. In this section, the frequency is estimated after synthesis[3] and implementation[4].

Table 4. Implementations of KHAZAD.

Type	Nbr of LUT	Nbr of registers	Nbr of slices	Latency (cycles)	Output every (cycles)	Freq. after Synt. (Mhz)	Freq. after Impl. (Mhz)
Fast KHAZAD	11328	13568	8800	62	1	175	148
Low area KHAZAD	11072	9600	7175	53	1	137	123

[2] FPGA Express (SYNOPSYS).
[3] FPGA Express (SYNOPSYS).
[4] Xilinx ISE4.

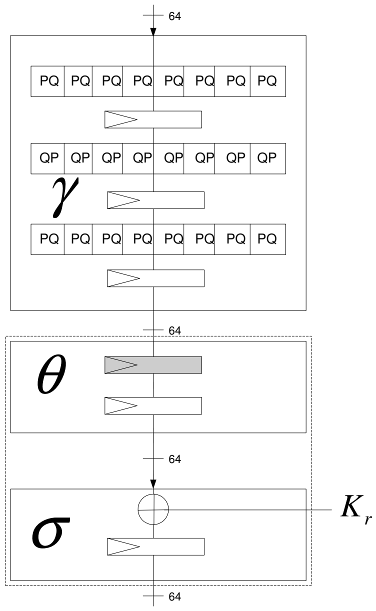

Fig. 5. The KHAZAD round function ρ.

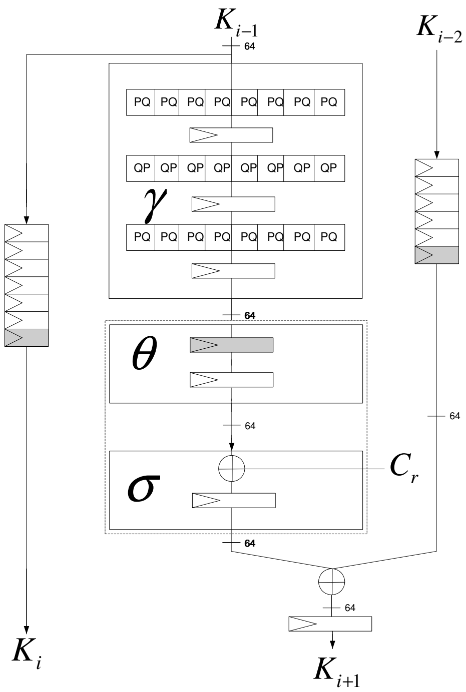

Fig. 6. The KHAZAD keyround.

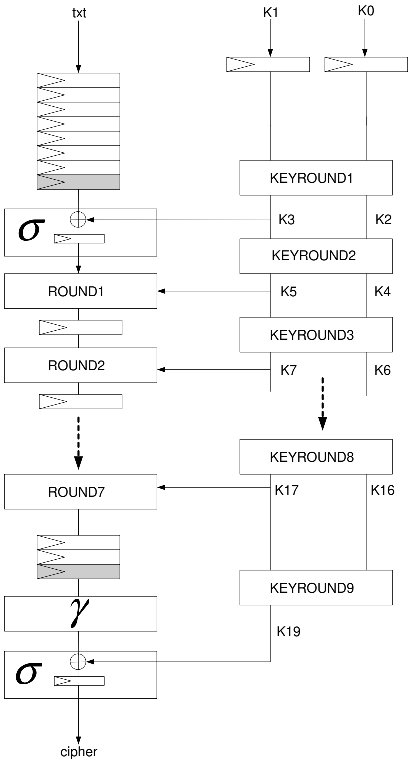

Fig. 7. KHAZAD.

From these results, we observe very high frequencies after synthesis. However critical delays mainly occurs when trying to place and route these synthesis results. The resulting implemented designs have surprising critical paths including 20% of logic and 80% of routes. We conclude that the real bottleneck of such large ciphers is in the difficulty of having an efficient place and route. Actually, constraints come from shift registers and high fanout. Implementation could probably be improved by replacing the last stage of shift registers by flip-flops, but the additional degree of freedom for the routes would be balanced with additional resources. Anyway, the resulting designs are very efficient as we will underline in section 7.

5 Block cipher description: MISTY1

MISTY1 is an iterated block cipher that operates on a 64-bit block with a 128-bit key and with a variable number of rounds n. We describe the algorithm with $n = 8$, as recommended in [4,5]. In the following subsections, we describe the data randomizing part and the key scheduling part of MISTY1 with their different components.

5.1 Data randomizing part

Figure 9 shows the data randomizing part of MISTY1[5]. The 64-bit plaintext P is divided in two 32-bit parts. Both parts are transformed into the 64-bit ciphertext using bitwise XOR operations, sub-functions FO_i $(1 \leq i \leq (n = 8))$ and sub-functions FL_i $(1 \leq i \leq (n + 2 = 10))$.

***FO_i* function.** Figure 8 shows the structure of FO_i[6]. This function split the input into two 16-bit strings. Then, it transforms both strings into the output with bitwise XOR operations and sub-functions FI_{ij} $(1 \leq j \leq 3)$. KO_{ij} $(1 \leq j \leq 4)$ and KI_{ij} $(1 \leq j \leq 3)$ are the left j-th 16 bits of KO_i and KI_i, respectively.

***FI_{ij}* function.** Figure 8 also shows the structure of FI_{ij}. The input is divided into two parts: a 9-bit string and a 7-bit string. These strings are transformed into the output using bitwise XOR operations and substitutions tables S_7 and S_9. In the beginning and the end of FI_{ij} function, the 7-bit string is zero-extend to 9 bits, and in the middle part, the 9-bit string is truncated to 7 bits eliminating its highest two bits (MSB). KI_{ij1} and KI_{ij2} are the left 7 bits and the right 9 bits of KI_{ij}, respectively.

***FL_i* function.** The structure of FL_i function is illustrated on figure 8. The 32-bit input is divided into two equal parts. The function transforms both parts into the output with bitwise AND, OR and XOR operations. KL_{ij1} $(1 \leq j \leq 2)$ is the left j-th 16 bits of KL_i.

S_7 and S_9 substitution functions. For the selection of S_7 and S_9 substitution functions, Matsui considers three criteria:

1. Their average differential/linear probability must be minimal,
2. Their delay time in hardware is as short as possible,
3. Their algebraic degree is high, if possible.

Based on these criteria, for the S_7 substitution function, Matsui chooses the following mathematical description:

$y_0 = x_0 + x_1x_3 + x_0x_3x_4 + x_1x_5 + x_0x_2x_5 + x_4x_5 + x_0x_1x_6 + x_2x_6 + x_0x_5x_6 + x_3x_5x_6 + 1$
$y_1 = x_0x_2 + x_0x_4 + x_3x_4 + x_1x_5 + x_2x_4x_5 + x_6 + x_0x_6 + x_3x_6 + x_2x_3x_6 + x_1x_4x_6 + x_0x_5x_6 + 1$
$y_2 = x_1x_2 + x_0x_2x_3 + x_4 + x_1x_4 + x_0x_1x_4 + x_0x_5 + x_0x_4x_5 + x_3x_4x_5 + x_1x_6x_3x_6 +$
$x_0x_3x_6 + x_4x_6 + x_2x_4x_6$
$y_3 = x_0 + x_1 + x_0x_1x_2 + x_0x_3 + x_2x_4 + x_1x_4x_5 + x_2x_6 + x_1x_3x_6 + x_0x_4x_6 + x_5x_6 + 1$
$y_4 = x_2x_3 + x_0x_4 + x_1x_3x_4 + x_5 + x_2x_5 + x_1x_2x_5 + x_0x_3x_5 + x_1x_6 + x_1x_5x_6 + x_4x_5x_6 + 1$
$y_5 =$
$x_0 + x_1 + x_2 + x_0x_1x_2 + x_0x_3 + x_1x_2x_3 + x_1x_4 + x_0x_2x_4 + x_0x_5 + x_0x_1x_5 + x_3x_5 + x_0x_6 + x_2x_5x_6$
$y_6 = x_0x_1 + x_3 + x_0x_3 + x_2x_3x_4 + x_0x_5 + x_2x_5 + x_3x_5 + x_1x_3x_5 + x_1x_6 + x_1x_2x_6 +$
$x_0x_3x_6 + x_4x_6 + x_2x_5x_6$

Based on the same above criteria, the S_9 function is defined as:

[5] Where registers needed for efficiency purposes are already mentioned.
[6] Where registers needed for efficiency purposes are already mentioned.

$$y_0 = x_0x_4 + x_0x_5 + x_1x_5 + x_1x_6 + x_2x_6 + x_2x_7 + x_3x_7 + x_3x_8 + x_4x_8 + 1$$
$$y_1 = x_0x_2 + x_3 + x_1x_3 + x_2x_3 + x_3x_4 + x_4x_5 + x_0x_6 + x_2x_6 + x_7 + x_0x_8 + x_3x_8 + x_5x_8 + 1$$
$$y_2 = x_0x_1 + x_1x_3 + x_4 + x_0x_4 + x_2x_4 + x_3x_4 + x_4x_5 + x_0x_6 + x_5x_6 + x_1x_7 + x_3x_7 + x_8$$
$$y_3 = x_0 + x_1x_2 + x_2x_4 + x_5 + x_1x_5 + x_3x_5 + x_4x_5 + x_5x_6 + x_1x_7 + x_6x_7 + x_2x_8 + x_4x_8$$
$$y_4 = x_1 + x_0x_3 + x_2x_3 + x_0x_5 + x_3x_5 + x_6 + x_2x_6 + x_4x_6 + x_5x_6 + x_6x_7 + x_2x_8 + x_7x_8$$
$$y_5 = x_2 + x_0x_3 + x_1x_4 + x_3x_4 + x_1x_6 + x_4x_6 + x_7 + x_3x_7 + x_5x_7 + x_6x_7 + x_0x_8 + x_7x_8$$
$$y_6 = x_0x_1 + x_3 + x_1x_4 + x_2x_5 + x_4x_5 + x_2x_7 + x_5x_7 + x_8 + x_0x_8 + x_4x_8 + x_6x_8 + x_7x_8 + 1$$
$$y_7 = x_1 + x_0x_1 + x_1x_2 + x_2x_3 + x_0x_4 + x_5 + x_1x_6 + x_3x_6 + x_0x_7 + x_4x_7 + x_6x_7 + x_1x_8 + 1$$
$$y_8 = x_0 + x_0x_1 + x_1x_2 + x_4 + x_0x_5 + x_2x_5 + x_3x_6 + x_5x_6 + x_0x_7 + x_0x_8 + x_3x_8 + x_6x_8 + 1$$

Both substitution boxes are defined as ROM tables in [4]. To optimize the number of logic cells used in FPGA implementations, we prefer to implement S_7 and S_9 functions directly as logical expressions. With enough pipelined stages, we keep the critical path of the design under control.

5.2 Key scheduling part

Figure 10 shows the key scheduling part of MISTY1[7]. K_i ($1 \leq$ i ≤ 8) is the left i-th 16 bits of the secret input key K. $K_i^{'}$ ($1 \leq i \leq 8$) corresponds to the output of FI_{ij} where the input of FI_{ij} is assigned to K_i and the key KI_{ij} is set to $K_{(i+1)mod 8}$.

The assignment between key scheduling subkeys K_i/ $K_i^{'}$ and the round subkeys $KO_{ij}, KI_{ij}, KL_{ij}$ is defined as follows, where i equals to $(i-8)$ when $(i > 8)$:

Table 5. Subkeys distribution.

Encrypt Round	KO_{i1}	KO_{i2}	KO_{i3}	KO_{i4}	KI_{i1}	KI_{i2}	KI_{i3}	KL_{i1}	KL_{i2}
Key round	K_i	K_{i+2}	K_{i+7}	K_{i+4}	$K_{i+5}^{'}$	$K_{i+1}^{'}$	$K_{i+3}^{'}$	$K_{\frac{i+1}{2}}(odd.i)$ $K_{\frac{i}{2}+1}^{'}(even.i)$	$K_{\frac{i+1}{2}+6}^{'}(odd.i)$ $K_{\frac{i}{2}+4}(even.i)$

This concludes the mathematical description of MISTY1 algorithm. The next section explains our FPGA design choices in order to be efficient in term of speed and resources used.

6 Implementation: MISTY1

In order to achieve the fastest FPGA implementation of MISTY1, we decided to limit the critical path to only one 4-input LUT and routes. Consequently, we do not use additional XOR's and multiplexors F5, F6 available in the VIRTEX slice. However, these functions could be used in order to reduce the area requirements of MISTY1.

Based on this delay constraint, we modified the mathematical description of the algorithm in order to regroup a maximum number of functions in a minimum number of 4-input LUT's. This strategy leads to very fast designs.

[7] Where registers needed for efficiency purposes are already mentioned.

6.1 S_7 and S_9 implementations

For S_7 and S_9 implementations, we used the logical expressions in place of substitution tables in order to reduce the number of logic cells used. The logical functions have to be pipelined in order to limit the critical path to only one 4-input LUT and routes. For S_7 and S_9, we designed two 2-stage pipelined versions. The next table shows the results that we obtained after synthesis:

Table 6. MISTY1: S_7 and S_9 synthesis results.

Component	Nbr of LUT's	Nbr of FF's	Nbr of pipelined stages
S_7	45	45	2
S_9	44	35	2

6.2 FO_i, FI_{ij}, FL_i implementations

Figure 8 details how we implemented FO_i, FI_{ij}, FL_i functions in order to limit the critical path to only one 4-input LUT and routes. As mentioned on the figure, we have to put an additional output pipeline stage into FI_{ij} function in order to correspond with the key scheduling part. The next table shows the results that we obtained after synthesis:

Table 7. MISTY1: FO_i, FI_{ij}, FL_i synthesis results.

Component	Nbr of LUT's	Nbr of FF's	Nbr of pipelined stages
FO_i	565	633	24
FI_{ij}	158	195	7
FL_i	32	32	1

6.3 The data randomizing part of MISTY1

For the same delay constraints, we obtained the design detailed in figure 9. Additional registers for input and output bits are needed to increase the speed performances and these are packed into IOBs. We finally get a 208-stage pipelined design.

6.4 The key scheduling part of MISTY1

Figure 10 shows the key scheduling part of MISTY1. Additional registers for input key bits are also packed into IOBs in order to increase performances. The assignment between key scheduling subkeys K_i/ $K_i^{'}$ and the round subkeys KO_{ij}, KI_{ij}, KL_{ij} is defined in table 5. We do the same in hardware putting the correct

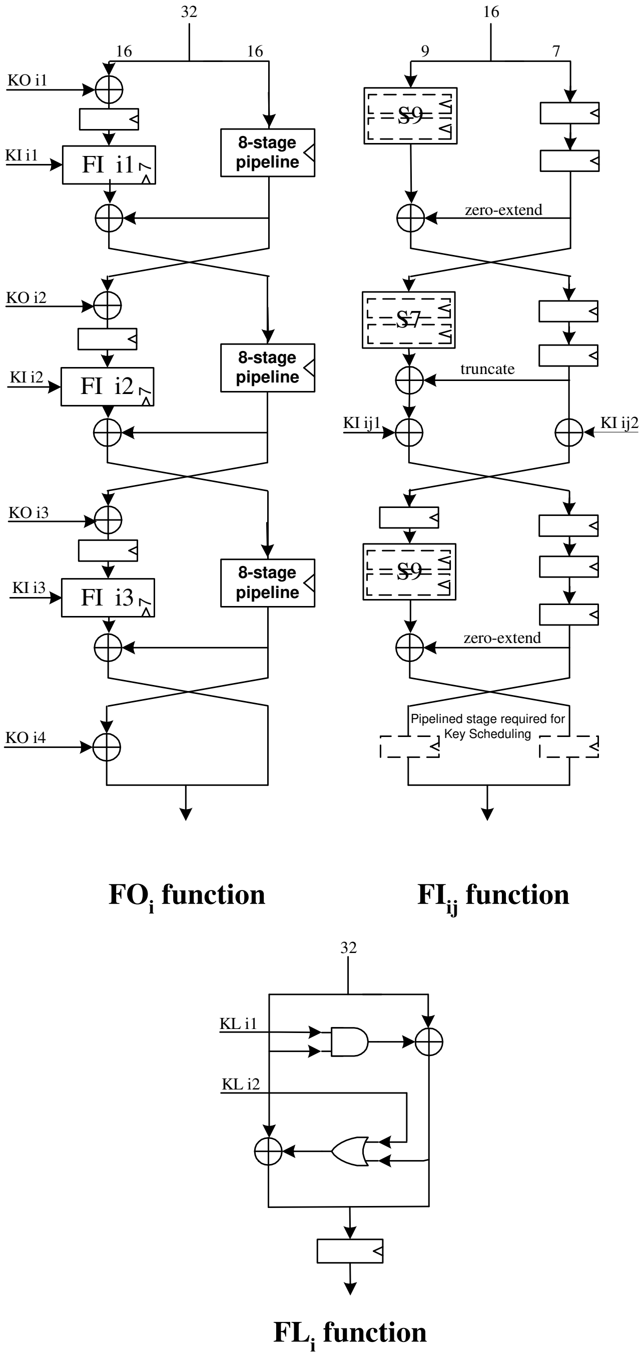

Fig. 8. MISTY1: FO_i, FI_{ij}, FL_i hardware designs.

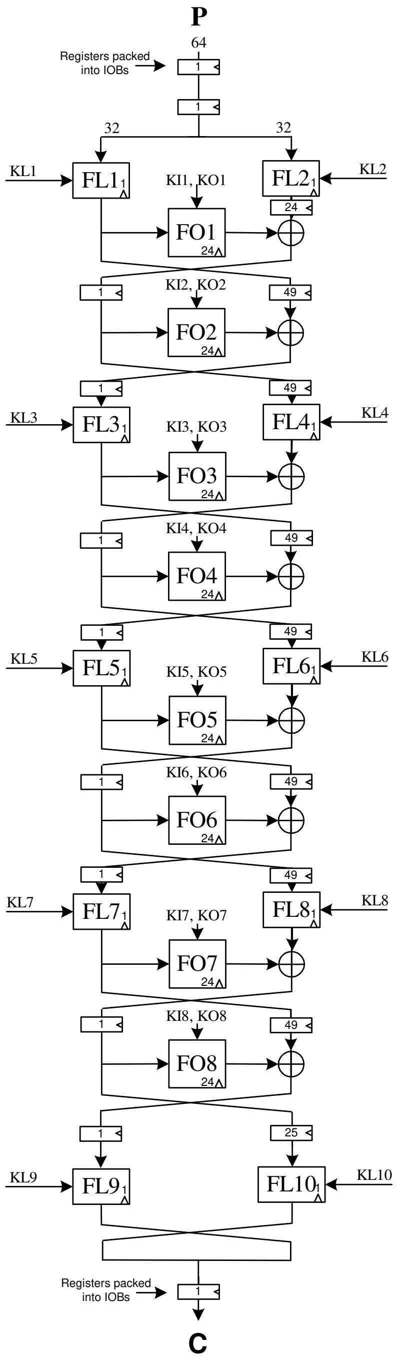

Fig. 9. The data randomizing part of MISTY1.

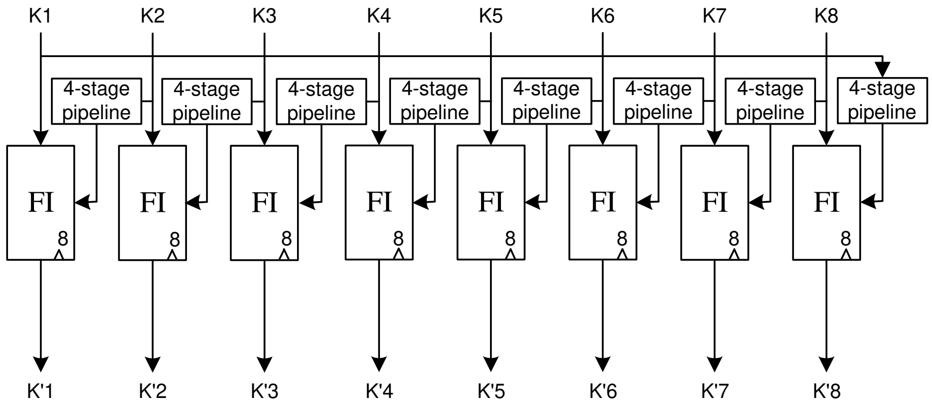

Fig. 10. MISTY1: Key Scheduling.

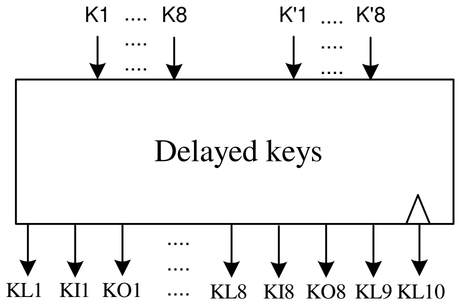

Fig. 11. MISTY1: Subkeys distribution.

number of pipelined stages for every round subkeys. Therefore, to achieve the key distribution, we use 16-bit shift registers, every one fitting in one LUT. Figure 11 represents the subkeys distribution.

Table 8 summarizes the result that we obtained for the complete key scheduling part.

Table 8. MISTY1: Key scheduling synthesis results.

Nbr of LUT's	Nbr of FF's	Nbr of pipelined stages
4912	2352	208

6.5 The complete cipher

The complete MISTY1 combines the data randomizing part and the key scheduling part from the precedent descriptions. Our results are summarized in table 9 where the latency is the number of pipelined stages. We propose a post-map[8]

[8] XILINX ISE4.

and a post-implementation[9] estimated frequency. The second one takes the routing delays into account. From this result, we observe a very high frequency after

Table 9. Implementation of MISTY1.

Nbr of LUT	Nbr of registers	Nbr of slices	Latency (cycles)	Output every (cycles)	Frequency (MHz) Post-map	Frequency (MHz) Post-implementation
10920	8480	8386	208	1	204	140

mapping phase but the final design only runs at 140 MHz. As we observed for KHAZAD, critical delays are mainly caused by the routing task. Because the critical path was limited to one 4-input LUT, the problem is even more critical than for KHAZAD. We conclude that:

1. It is not so easy to deal with routing delays. They are no systematic tools to prevent these ones. All we can do, is to locate the problem and to redesign the global circuit. Nevertheless, routing problems usually come from shift registers and high fanout. Implementation could probably be improved by replacing the last stage of shift registers by flip-flops.
2. Trying to reduce the critical path to only one 4-input LUT is not always the best choice. Indeed, if a big part of the critical path is due to route, the use of additional XORs, F5,F6 can reduce the number of logic cells used without increasing the critical path.

Anyway, the resulting design is very efficient and suitable for FPGA, as proved in the next section.

7 Comparison with AES RIJNDAEL, SERPENT and MISTY1

In order to evaluate our implementation results and the hardware suitability of KHAZAD and MISTY1, we compare them with similar results obtained with the Advanced Encryption Standard RIJNDAEL and SERPENT [2]. We chose RIJNDAEL because of its status of new encryption standard and SERPENT because it seems that it was the best AES candidate regarding FPGA implementations. However, comparisons between KHAZAD and MISTY1 seem to be more relevant because they were implemented using the same methodology.
In [2], the Xilinx VIRTEX1000BG560-4 was selected as the target device for evaluation of AES candidates. Table 10 compares RIJNDAEL, SERPENT, MISTY1 and KHAZAD encryption circuits in terms of hardware cost, frequency and throughput for VIRTEX1000bg560-6 component. The hardware cost in LUT and registers is replaced by a number of slices. We also investigate the ratio Throughput/Area which is a good measurement of hardware efficiency.

[9] XILINX ISE4.

Table 10. Comparisons with RIJNDAEL, SERPENT and MISTY1.

Type	Nbr of slices	Output every (clk edges)	Estimated frequency(Mhz)	Throughput (Mbits/s)	Throughput/Area ($\frac{Mbits/s}{slices}$)
RIJNDAEL [2]	10992	2.1	31.8	1938	0.18
SERPENT [2]	9004	1	38	4860	0.54
MISTY1	8386	1	140	8960	1.07
Fast KHAZAD	8800	1	148	9472	1.07
Low area KHAZAD	7175	1	123	7872	1.09

8 Conclusions

We propose efficient FPGA implementations of block ciphers KHAZAD and MISTY1. The structure of KHAZAD is very close to AES RIJNDAEL and offers comparable security. However, improvements have been done concerning implementation aspects and these allow very efficient FPGA implementations for high throughput applications. Although its keyround is still very expensive, KHAZAD is a very suitable block cipher for FPGA implementation in the context described for these experiments. MISTY1 offers similar performances and its main bottleneck is to be found in the routing delays. By avoiding these problems, we could greatly improve the design frequency.

Upon comparison, our implementations of KHAZAD and MISTY1 offer better results than those reported for RIJNDAEL and SERPENT in [2], but the implementation of RIJNDAEL with the design methodology described in this paper will deserve a forthcoming work and allow more relevant comparisons.

Concerning KHAZAD and MISTY1, we believe that both ciphers have interesting properties for hardware implementations. MISTY1 offers slight advantages in terms of hardware cost: it has the Feistel structure and low-cost substitution boxes. Its key scheduling is also less expansive than KHAZAD. However, the intensive use of shift registers to pipeline the Feistel network makes the algorithm structure more complex and the design more difficult to route. It results in a larger latency. Looking at the final results, the ratio $Throughput/Area$ illustrates that both ciphers are very close and sufficiently efficient but potential improvements exist for MISTY1. It seems that reconfigurable hardware implementations will not be the bottleneck for the selection of KHAZAD or MISTY1 as a NESSIE cipher.

References

1. P. S. L. M. Barreto and V. Rijmen, "The KHAZAD legacy-level block cipher." Primitive submitted to NESSIE, Sept. 2000.
2. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists." in *Proceedings of the Third Advanced Encryption Standard Conference*, pp. 13–27, NIST, Apr. 2000.
3. K. Gaj and P. Chodowiec, "Comparison of the hardware performance of the AES candidates using reconfigurable hardware." in *Proceedings of the Third Advanced Encryption Standard Conference*, pp. 40–54, NIST, Apr. 2000.

4. M. Matsui, “Block encryption algorithm MISTY.” in *Proceedings of Fast Software Encryption – FSE’97* (E. Biham, ed.), no. 1267 in Lecture Notes in Computer Science, pp. 64–74, Springer-Verlag, 1997.
5. M. Matsui, “Specification of MISTY1 - a 64-bit block cipher.” Primitive submitted to NESSIE by E. Takeda, Mitsubishi, Sept. 2000.
6. M. McLoone and J. V. McCanny, “High performance single-chip FPGA rijndael algorithm implementations.” in *Proceedings of CHES’01* (Çetin Kaya Koç, D. Naccache, and C. Paar, eds.), no. 2162 in Lecture Notes in Computer Science, pp. 65–76, Springer-Verlag, 2001.
7. Xilinx, *Virtex 2.5V Field Programmable Gate Arrays Data Sheet*. URL: `http://www.xilinx.com`.