Journal of Universal Computer Science, vol. 13, no. 3 (2007), 407-418
submitted: 27/11/06, accepted: 12/3/07, appeared: 28/3/07 © *J.UCS*

A Dynamically and Partially Reconfigurable Implementation of the IDEA Algorithm Using FPGAs and Handel-C

José M. Granado-Criado, Miguel A. Vega-Rodríguez,
Juan M. Sánchez-Pérez, Juan A. Gómez-Pulido
(Dept. of Computer Science, Univ. of Extremadura
Escuela Politécnica, Campus Universitario s/n. 10071 Cáceres, Spain
{granado, mavega, sanperez, jangomez}@unex.es)

Abstract: Nowadays, the information security has achieved a great importance, both when information is sent through a non-secure network (as the Internet) and when data are stored in massive storage devices. The cryptographic algorithms are used in order to guarantee the security of data sent or stored. A lot of research is being done in order to improve the performance of the current cryptographic algorithms, including the use of FPGAs. In this work we present an implementation of the IDEA cryptographic algorithm using reconfigurable hardware (FPGAs). In addition, in order to improve the performance of the algorithm, partial and dynamic reconfiguration has been used to implement our final circuit. This fact allows us to obtain a very high encryption speed (14.757 Gb/s), getting better results than those found in the literature.

Keywords: FPGA, Partial and Dynamic Reconfiguration, IDEA, Cryptography.
Categories: E.3, C.4

1 Introduction

At present, the field of cryptography is on the increase in the telecommunications world. As a consequence, there is a constant need for more secure and efficient cryptographic algorithms. Among them we find the IDEA algorithm [Schneier, 96], the one used in our research. This is one of the most popular algorithms, especially thanks to its use in the PGP (Pretty Good Privacy) system [Garfinkel, 95].

We have made an implementation of the IDEA cryptographic algorithm using FPGAs and both partial and dynamic reconfiguration. Other authors have already done implementations of this algorithm using FPGAs; among them we can point out [González, 03], where KCM multipliers and a super-pipelining implementation are used to achieve a high performance implementation (8.3 Gb/s). We use a similar technique, but we develop new reconfigurable elements (like KCA) and we combine the utilization of VHDL to implement the reconfigurable elements and Handel-C [Celoxica, 05] to implement the non-reconfigurable elements. These improvements give us a better performance (14.757 Gb/s). In addition, Handel-C language allows us to decrease the development time because it is a high-level language closer to the traditional programmer. Pipelining is also used in other papers, like [Hämäläinen, 02], who implements 7 stages by phase, whereas we employ 16 stages. The optimization of the multipliers is also an important task. In [Hämäläinen, 02] partial product

generation and a three-stage diminished-one adder is used to calculate the multiplication modulo $2^{16}+1$. We can find another multiplier implementation technique in [Leong, 00] and [Cheung, 01], where parallel-serial multipliers are used. However, we implement KCM multipliers [Vaidyanathan, 03], that is, constant coefficient multipliers which are very fast because they employ LUTs to store part of the result of the multiplication.

Taking all this into account, we achieve to improve the results obtained by these authors, among which the best result is 8.3 Gb/s [González, 03] and our result is 14.757 Gb/s.

The structure of this work is the following: First of all, we present a description of the IDEA cryptographic algorithm; next we show the languages used to implement the components. We also provide a description of the implemented (dynamically and not dynamically reconfigurable) components. Later, we describe how the partial and dynamic reconfiguration is performed, showing the steps followed to implement the IDEA algorithm. Finally, we analyse the results and give the conclusions.

2 The IDEA algorithm

IDEA [Schneier, 96] is a 64-bit text block cryptographic algorithm which uses a 128-bit key. This key is the same for both encryption and decryption, in other words, it is a symmetric algorithm, and it is used to generate 52 16-bit keys.

The algorithm consists of 9 phases: 8 identical phases (figure 1(a)) and a final transformation phase (figure 1(b)). The encryption takes place when the 64-bit block is propagated through each of the first 8 phases in a serial form, where the block, divided into four 16-bit sub-blocks, is modified using the six sub-keys corresponding to each phase (elements Z_j^i of figure 1: 6 sub-keys per phase and 4 sub-keys for the last phase). When the output of the 8^{th} phase is obtained, the block goes through a last phase, the transformation one, which uses the last 4 sub-keys.

The decryption follows an identical pattern but computing the sum or multiplication inverse of each sub-key, depending on the case, and altering the order of use of the sub-keys.

As we can suppose, the major problem lies in the multipliers, since, apart from taking a great amount of computation and resources, they are executed 4 times in each phase. The improvement of this component is one of the most studied aspects in the literature. In our case, we will use KCM multipliers and partial and dynamic reconfiguration, as we will see later.

3 Hardware description languages used

We have used two hardware description languages to implement the components:

First of all, we have used the VHDL language [Pedroni, 04] to implement components which will be reconfigured in runtime.

We have also used the Handel-C language [Celoxica, 05] of Celoxica [Celoxica, 07], a language with an abstraction level which is higher than VHDL. We employ this language to implement the pipelining registers, the memory, the interconnection

among the implemented components using VHDL and the Host-FPGA communication.

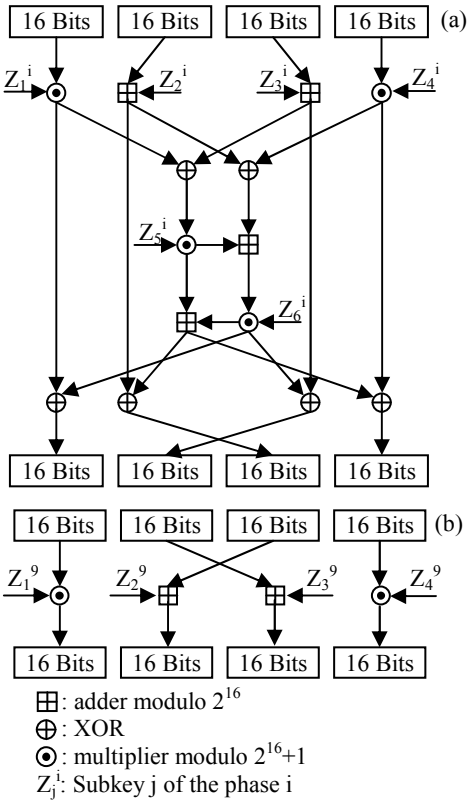

Figure 1: Description of the cryptographic IDEA algorithm phases.

4 Reconfigurable elements of the design

In figures 1(a) and 1(b) all multipliers and several adders make their operations with a sub-key which will be fixed in the complete encryption process.

This allows us to use KCM (Constant Coefficient Multiplier) multipliers and KCA (Constant Coefficient Adder) adders, which will be changed dynamically in runtime.

4.1 KCM multipliers

KCMs [Vaidyanathan, 03] are a kind of multiplier in which a datum is a constant. They are based on the use of LUTs (Look-Up Table) to store all possible multiplication values. In order to avoid storing the possible 2^{16} results of the multiplication (we have 16-bit data), we have chosen the following procedure.

We use only 4 LUTs with 4 input bits and 20 output bits (the resultant number of bits of multiplying a 4-bit number by a 16-bit number).

Then, we built an adder tree using the LUT results in order to obtain the final multiplication result (figure 2). Each LUT of the KCM multiplier is composed of 20 4x1 LUTs . Each 4x1 LUT obtains one bit of the multiplication result. The content of these 4x1 LUTs will be changed in runtime according to the value of the subkey used in each multiplication.

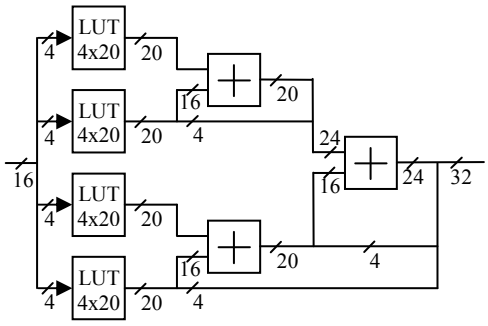

Figure 2: KCM multiplier structure.

4.2 The KCM multiplier modulo $2^{16}+1$

The multipliers used in the description of the IDEA algorithm phases are multipliers modulo $2^{16}+1$. In order to implement these multipliers we use the Low-High algorithm [Lai, 92]. In this algorithm we can distinguish three situations:

- a==0 ➔ The result will be 1-k.
- k==0 ➔ The result will be 1-a.
- a≠0 and k≠0 ➔ The result will be a*k in modulo $2^{16}+1$.

Where 'a' stands for the input datum of the multiplier and 'k' stands for the sub-key of the corresponding phase and multiplier.

Let see how to implement the algorithm.

4.2.1 The $2^{16}+1$ multiplication

In order to carry out the $2^{16}+1$ multiplications, we do the following operations:

1. m=a*k (using the KCM multiplier)
2. r=m[15:0]-m[31:16]
3. If (m[15:0]<m[31:16]) ➔ mult$2^{16}+1$=r+1
   Else ➔ mult$2^{16}+1$=r

The resulting circuit is shown in figure 3.

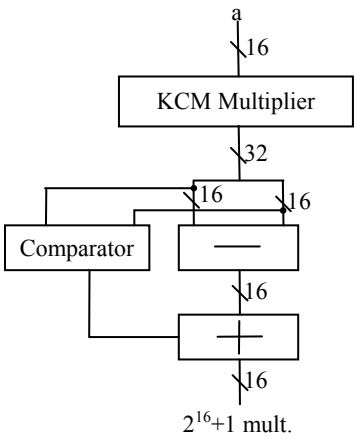

Figure 3: 2^{16}+1 multiplier structure.

4.2.2 Multiplication and subtractions

In the Low-High algorithm the *1-k* and *1-a* subtractions are done, that is, we need two subtractors. Nevertheless, as *k* is a constant in this case, then *1-k* is a constant too. So, the *1-k* calculation is done in a reconfigurable way.

On the other hand, in the original algorithm a multiplexer selects either *a*k*, *1-k* or *1-a*. However, in this case, we employ a reconfigurable method to select the output. This is possible because, in runtime, we know if *k* is zero or not. Therefore, we have two possibilities:

- if k==0 then the output = 1-a.
- if $k \neq 0$ the output depends on the a value (=0 or ≠0).

The first case is easy: if k==0 the selector circuit will let pass the value *1-a*.

1-k	a*k	ZeroComp[1]	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 1: Truth table of the i^{th}-bit selection of the complete KCM multiplier modulo 2^{16}+1 for $k \neq 0$.

[1] This value indicates if a==0 (value 0) or if a≠0 (value 1).

In the second case, we do not know the *a* value (because it is variable) in the partial and dynamic reconfiguration time. So, for any *a* value we must make the selector circuit to choose the right value. To do that, we are going to analyze the truth table shown in table 1.

As we know, when the dynamic and partial reconfiguration of the selector circuit is done, we do not know the value of *a*. So, we will obtain the output based on the *1-k* value because it is the only value we know. Therefore, we obtain that:

- If *1-k*==0 then the output = *ZeroComp* & *a*k*
- If *1-k*≠0 then the output = ~*ZeroComp* | (*ZeroComp* & *a*k*)

Therefore, the complete code for each selection bit is the following one:

```
If (k==0)
      Output=1-a
Else
      If (1-k==0)
            Output=ZeroComp & k*a
      Else
            Output=~ZeroComp | (ZeroComp & k*a)
```

Since we have based the output calculation of the selector circuit on the *1-k* value, which is the only one known before the encryption process, we can generate the selector circuit by means of dynamic and partial reconfiguration.

So, to implement the selector circuit, we employ 16 3x1 LUTs (one for each bit) which will be reconfigured in runtime following the previous code. This will be done in order for each LUT output to be calculated in function of its inputs, which are the *a*k, 1-a* and *ZeroComp* values.

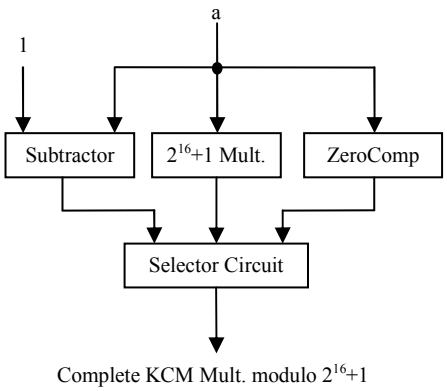

Figure 4: Complete selector circuit of the KCM multiplier modulo $2^{16}+1$.

In figure 4 the full circuit of the KCM multiplier modulo $2^{16}+1$ is shown. In addition, it is important to mention that the circuit described in table 1 also obtains the *1-k* value, in case the value of *a* is zero.

4.3 KCA adders

The KCA adders (Constant Coefficient Adders) are based on the same idea than the KCM multipliers. Three 4x5 LUTs and one 4x4 LUT are used to store the values resulting from adding any input value to a constant.

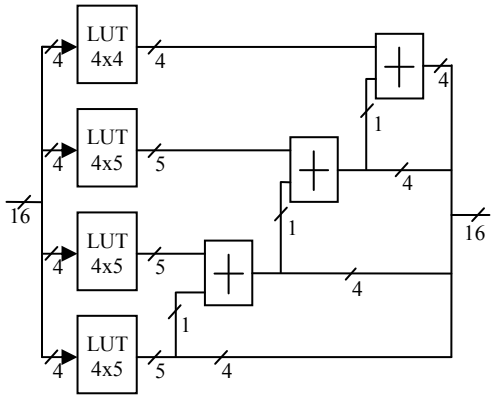

Figure 5: KCA adder structure.

The difference with the KCM multiplier is that in the KCM multiplier each 4x20 LUT stores the result of multiplying a 4-bit value by a 16-bit constant. On the other hand, in the case of the adder, the additions are made in groups of 4 bits, that is, the 4 LSB of the constant will be added to the 4 LSB of the *a* value, repeating the process with the other three groups, considering, of course, the carries. The carries are the elements which explain the 5^{th} output bit of the 4x5 LUTs: when two 4-bit values are added, we will have a 5-bit result at most, the carry being the most significant bit. However, the most significant 4x4 LUT does not use the carry bit because these components are adders modulo 2^{16}, this is the reason why that bit is despised.

At the end, as happened with the KCM multiplier, the LUTs results will be the inputs of an adder tree whose output will be the KCA adder final result. The complete design of this element can be seen in figure 5.

5 Not reconfigurable elements

Apart from the already mentioned components, we have implemented a set of components using Handel-C which is not reconfigured in runtime. These components are:

- A dual-port RAM memory: We have implemented a 1024-word RAM memory with a 64-bit word wide, that is, the block size, in order to store the data that are going to be processed. The reason why we need this memory is to make loads and storages simultaneously.

- Pipelining registers: We have also implemented a total of 806 16-bit registers, 34 32-bit registers and 102 1-bit registers to store the data in between the pipeline stages. These registers can be seen in figures 6 and 7.
- Host-FPGA communication system: The PCI bus is used to communicate the FPGA with the Host application.

6 Performing the partial and dynamic reconfiguration

We have used JBits [Sun, 04], which is a Java library designed to make the dynamic and partial reconfiguration.

We have implemented two functions: one to reconfigure the multipliers (including the selector circuit) and the other to reconfigure the adders. These two functions are used by another function which is in charge of making the reconfiguration of a complete phase. On the one hand, the first two functions receive the coordinates of the bottom left Slice of each component together with the corresponding sub-key. The third function, on the other hand, receives the bottom left coordinates of the phase which is going to be reconfigured. With these values, the three functions will be able to make the reconfiguration correctly.

To reconfigure the LUTs involved in the multiplications and additions we only need to call the function *jbits.setCLBBits* with the results of the corresponding operation for the received constant.

However, for the selector circuit reconfiguration, we must employ the formulas of section 4.2. Thus, we obtain the following JBits variables:

```
int[] g_operator = Expr.G_LUT("~G3");
int[] g_k_non_zero = Expr.G_LUT("~(~G2 | (G2 & G1))");
int[] g_k_zero=Expr.G_LUT("~(G2& G1)");
```

Therefore, we only need to implement the code described in section 4.2 using the previous variables in the correct places.

7 The final implementation

The final implementation of the IDEA algorithm has been made in a Xilinx Virtex-2 6000 FPGA [Xilinx, 07] included in a Celoxica ADMXRC2 board. We have used this FPGA because both it has a very large number of resources, particularly a total of 33792 Slices, and it allows us to make partial and dynamic reconfiguration. It is important to clarify that although the ADMXRC2 has 18x18 multipliers, we do not use them because the KCM multipliers already make all the multiplications needed.

The implementation is based on a pipelined execution. The pipeline consists of the following stages: 16 pipeline stages from each one of the eight normal phases (figure 6), 4 pipeline stages from the transformation phase and, finally, two more stages for loading/storing. All this adds up to a total of 134 pipeline stages. In addition, we want to note that each of the KCM multipliers modulo $2^{16}+1$ consists of 4 pipeline stages, as we can see in figure 7.

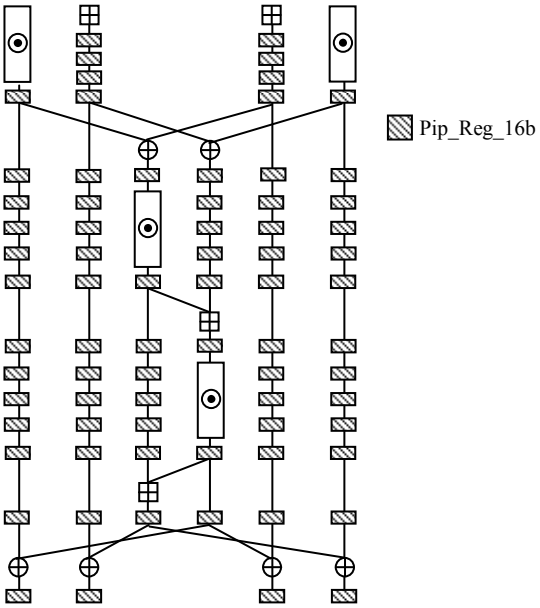

Figure 6: Pipeline of a phase of the IDEA algorithm.

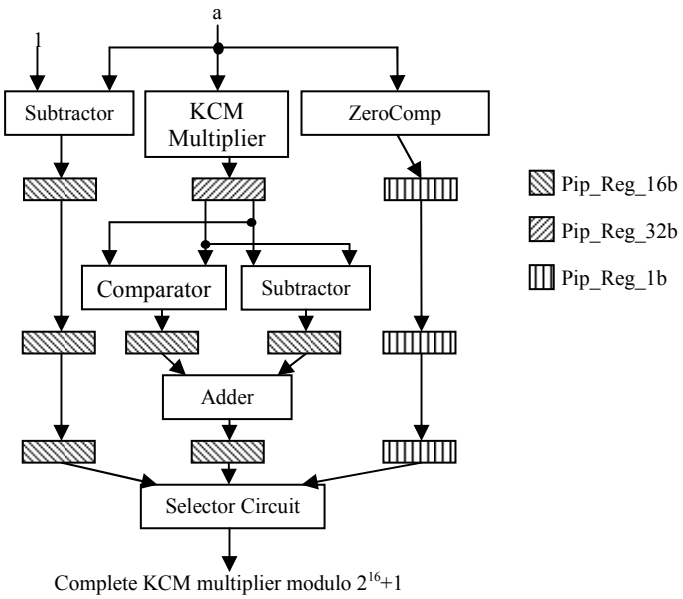

Figure 7: Pipeline of the KMC multiplier modulo $2^{16}+1$.

8 Results

As far as the results are concerned, we have made a total of ten empiric experiments in order to measure the performance of our implementation. The results from these experiments are shown in figure 8. As we can see, the results oscillate between 9.228 and 14.757 Gb/s, giving an average of 11.382 Gb/s. These values (including the least value obtained in this work) are better than the results achieved by other authors, as table 2 shows.

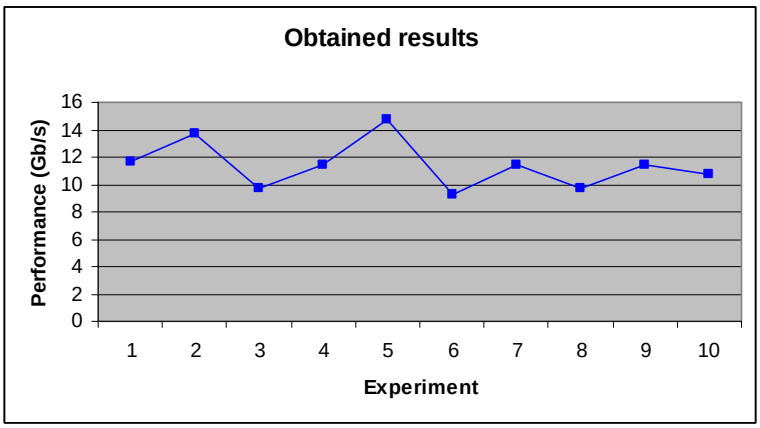

Figure 8: Obtained results by the implementation.

Device	Performance (Gb/s)	Reference
XC2V6000-6	11.4	This work
XCV600-6	8.3	[González, 03]
XCV1000-6	6.8	[Hämäläinen, 02]
XCV1000-6	5.1	[Cheung, 01]
XCV1000-4	4.3	[Beuchat, 02]
XCV1000-6	2.3	[Leong, 00]
XCV1000-6	1.5	[González, 02]

Table 2: Comparative results table.

Another interesting datum is the reconfiguration time. This includes the time needed to rewrite the configuration file (around 141 ms) and the time of the FPGA configuration (14.34 ms, because our implementation needs 11476 Slices, where one CLB includes 4 Slices, and the time to configure one CLB for Virtex-II FPGAs is 5 µs [Xilinx, 05a] [Xilinx, 05b]). This time may seem too high, but the reconfiguration is done when the user introduces the key and so he/she does not realise about this time (when the user push the *configure* button, the reconfiguration is already done).

9 Conclusions and future work

As a way of conclusion, it only remains to say that in this work we have presented an implementation of the IDEA cryptographic algorithm using an FPGA, Handel-C and dynamic and partial reconfiguration. This implementation is a pipelined version with 134 stages and uses constant coefficient multipliers and adders which are reconfigured in runtime in order to improve the performance. Thanks to this technique we have achieved an average performance of 11.382 Gb/s, reaching a peak of 14.757 Gb/s, improving the obtained results by others authors.

Therefore, we have proved that the use of partial and dynamic reconfiguration improves the performance of the IDEA algorithm by using KCM multipliers and KCA adders. In addition, although the super-pipelining increases the initial latency, it allows us to encrypt parts of a lot of data blocks at the same time and the clock frequency is increased substantially.

In the future, the next step will be the duplication of the data path in order to encrypt several blocks at the same time (for example, we will be able to encrypt blocks in the first path and to decrypt blocks in the second path without making a new FPGA configuration). This is possible because both the present implementation uses only 11476 Slices (33% of the FPGA resources) and the ECB mode of the IDEA algorithm is used, in which the encryption of one block is independent of the rest.

Acknowledgements

This work has been partially funded by the Ministry of Education and Science and FEDER under contract TIN2005-08818-C04-03 (the OPLINK project).

References

[Beuchat 02] Beuchat, J. L.; Haenni, J. O.; Teuscher, C.; Gómez, F. J.; Restrepo, H. F.; Sánchez, E.: "Approches Metérielles et Logicielles de lAlgorithme IDEA", Technique et Science Informatiques, vol. 21, no. 2, pp. 203-204, 2002.

[Celoxica, 05] Celoxica: "Handle-C Language Reference Manual". Version 3.1, 2005.

[Celoxica, 07] Celoxica: "http://www.celoxica.com", 2007.

[Cheung, 01] Cheung, O. Y. H.; Tsoi, K. H.; Leong, P. H. W.; Leong M. P.: "Tradeoffs in Parallel and Serial Implementations of the International Data Encryption Algorithm IDEA", Workshop on Cryptographic Hardware and Embedded Systems (CHES'2001), LNCS, Springer-Verlag, vol. 2162, pp. 333-347, Paris, France, May 2001.

[Garfinkel, 95] Garfinkel, S.: "PGP: Pretty Good Privacy". O'Reilly, 1995.

[González, 02] González, I.: "Codiseño en Sistemas Reconfigurables basado en Java", Internal Technical Report, UAM, Spain, December 2002.

[González, 03] González, I.; Lopez-Buedo, S.; Gómez, F. J.; Martínez, J.: "Using Partial Reconfiguration in Cryptographic Applications: An Implementation of the IDEA Algorithm", 13th International Conference on Field Programmable Logic and Application (FPL'2003), pp. 194-203, Lisbon, Portugal, September 2003.

[Hämäläinen, 02] Hämäläinen, A.; Tomminska, M.; Skittä, J.: "6.78 Gigabits per Second Implementation of the IDEA Cryptographic Algorithm", 12th International Conference on Field Programmable Logic and Application (FPL'2002), pp. 760-769, Montpellier, France, September 2002.

[Lai, 92] Lai, X.: "On the Design and Security of Block Ciphers", ETH Series in Information Processing, Konstanz, 1992.

[Leong, 00] Leong, M. P.; Cheung, O. Y. H.; Tsoi, K. H.; Leong, P. H. W.: "A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA", IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'2000), Napa Valley, CA, USA, pp. 122-131, April 2000.

[Pedroni, 04] Pedroni, V.A.: "Circuit Design with VHDL". The MIT Press, 2004.

[Schneier, 96] Schneier, B.: "Applied Cryptography". John Wiley & Sons, 2nd edition, 1996.

[Sun, 04] Sun Microsystems, "JBits User Guide", 2004.

[Vaidyanathan, 03] Vaidyanathan, R.; Trahan, J. L.: "Dynamic Reconfiguration: Architectures and Algorithms", Kluwer Academic/Plenum Publishers, 1st edition, 2003.

[Xilinx, 05a] Xilinx: "Virtex-II Platform FPGAs: Complete Data Sheet", 1 March 2005.

[Xilinx, 05b] Xilinx: "Virtex-II Platform User Guide". 23 March 2005.

[Xilinx, 07] Xilinx: "http://www.xilinx.com", 2007.