AES Embedded Hardware Implementation

Ould-cheikh Mourad, Si-Mohamed Lotfy
Labratoire d'Electronique, Ecole Polytechnique
BP17 Bordj-El-Bahri, Algeria
Email: farouk92003@hotmail.com

Mehallegue Noureddine, Bouridane Ahmed
School of Electronics Electrical Engineering
and Computer Science Queen's University
Belfast, BT7 1NN

Tanougast Camel
Laboratoire d'Instrumentation Electronique de Nancy (L.I.E.N)
Universit Henri Poincar - Facult des Sciences et Techniques
de Nancy 1 BP 239 Vandoeuvre Ls Nancy, France

Abstract

The paper presents a parallel reconfigurable hardware implementation of the AES cryptographic algorithm developed for an embedded application. This new methodology directly maps a design described in a high level language, Handel-C, to FPGA platforms. The Handel-C approach narrows the gap between performance and flexibility, and thus, reduce the risk of translating a high level prototype into HDLs. It provides a high degree of flexibility from two viewpoints: the language level of abstraction and the hardware reconfiguration. Using Handel-C, we enhanced the performance of the designed unit by applying parallelism and reconfigurability. Our FPGA implementations show that superior performance can be achieved compared with software and hardware implementations counterparts. In particular, our design outperforms most of the other designs in speed. At the same time, the area cost for putting the AES algorithm on the same hardware core is also kept as low as possible.

1. Introduction

Encryption has long been associated with military or government communication. However, the increasing use of electronic communications for commercial and private transactions has led to the universal use of encryption techniques to protect electronic communication against interception, theft, alteration or fraud.

Cryptographic systems can be classified into two classes: symmetric (secret key) and asymmetric (public key) cryptosystems. Symmetric cryptosystems, such as the Data Encryption Standard (DES), 3DES, and Advanced Encryption Standard (AES) use an identical key for both encryption and decryption processes. On the other hand, asymmetric cryptosystems, such as the Rivest-Shamir-Adleman (RSA) and Elliptic Curve algorithms, use different keys for the encryption and decryption tasks, thus eliminating the key management problem [1]. Symmetric cryptography is more suitable for the encryption of a large amount of data. The AES algorithm defined by the National Institute of Standards and Technology (NIST) of the United States has been widely accepted to replace DES as the new symmetric encryption algorithm [2] .

AES encryption is an efficient scheme for both hardware and software implementation. Currently reconfigurable computing is an intermediate solution between general purpose processors (GPPs) and application specific integrated circuits (ASICs). Because it employs hardware that is programmable by software, reconfigurable computing (RC) has advantages over both GPPs and ASICs. It provides a faster hardware solution than a GPP. Also, it has a wider applicability than ASICs since its configuring software makes use of the broad range of functionality supported by the reconfigurable device[3]. Field Programmable Gate Arrays (FPGAs), an instance of RCs, continue to grow in size and currently contain several millions of gates.

A typical embedded systems design methodology prototypes a target design in a high-level language, such as C, and then translates it manually into an HDL code. This process is time-consuming and error-prone [4]. Furthermore, for efficient designs, the details of target platforms should be known in advance when using HDLs. In other words, HDLs are lower level than high-level languages, which reduces the flexibility of a prototype.

In order to narrow the gap between performance and flexibility, reduce the time required to complete a design and reduce the risk of errors that might result from trans-

Second NASA/ESA Conference on Adaptive Hardware and Systems(AHS 2007)
0-7695-2866-X/07 $25.00 © 2007 IEEE

lating a high-level prototype into HDLs, a new design approach is emerging now-a-days, in which a high-level language is used for embedded system designs. There are different languages used for this purpose [5], and among the most emerging ones are Handel-C [6, 7] and SystemC [8, 9]. Other similar languages that can be used for embedded system designs include Carte, Catapult C, DIME-C, Impulse C, Mitrion C, Napa C, SA-C and Streams C. A comparison between these different languages can be found in Holland et al [5]. The focus in this paper is on the Handel-C design methodology.

Handel-C is a high level language that has the capability to be mapped directly to FPGAs. In addition, it has constructs, such as pipelining and parallelism that improve the performance of a design. Since it is not device-specific, Handel-C can synthesize a design to different FPGA chips, providing more flexibility and speed upgrading.

The Handel-C methodology, as used in this paper, allows for a high degree of flexibility from two viewpoints: the language level of abstraction and the hardware reconfiguration. Handel-C is like any other high level language that can be easily modified for updates and new standards. Moreover, it enables a design space exploration. FPGAs are reconfigurable and can easily be reprogrammed for new updates and standards. In addition to narrow the gap between performance and flexibility, this approach provides fast time-to-market (TTM).

This paper describes an embedded implementation of AES algorithm programmable hardware using Handel-C. This paper is organized as follows. AES algorithm is briefly described in Section 2. The architecture of the proposed AES design is described in Section 3. The application of the Handel-C design methodology to design this unit is discussed in Section 4. In Section 5, comparison with other similar works is discussed. Finally, we conclude the paper in Section 6.

2. The AES Algorithm: an Overview

The AES algorithm is asymmetric block cipher that processes data blocks of 128 bits using a cipher key of length 128, 192, or 256 bits. Each data block consists of a 4 4 array of bytes called the state, on which the basic operations of the AES algorithm are performed. The AES encryption/decryption procedure is shown in Figure 1. After an initial round key addition, a round function consisting of four different transformations - SubBytes, ShiftRows, MixColumns, and AddRoundKey - is applied to the data block (i.e., the state vector). The round function is performed iteratively 10, 12, or 14 times, depending on the key length of 128, 192 or 256, respectively [10, 11]. Note that in the last round MixColumns is not applied. The four transformations are described briefly as follows:

o SubBytes: a nonlinear byte substitution that operates independently on each byte of the state using a substitution table (the SBox)

o ShiftRows: a circular shifting operation on the rows of the state with different numbers of bytes (offsets)

o MixColumns: the operation that mixes the bytes in each column by the multiplication of the state with a fixed polynomial modulo $x^4 + 1$

o AddRoundKey: an XOR operation that adds a round key to the state in each iteration, where the round keys are generated during the key expansion phase

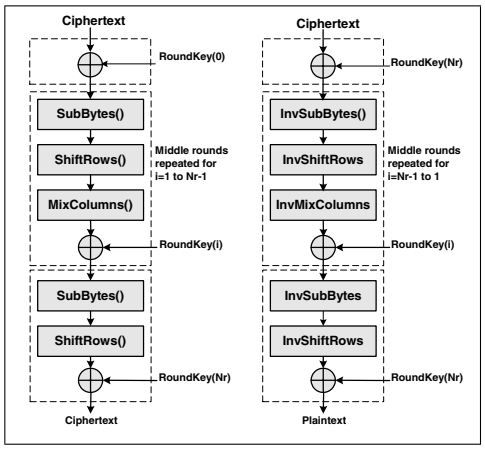

Figure 1. The AES encryption/decryption procedure.

The decryption procedure of the AES is basically the inverse of each transformation (InvSub-Bytes, InvShiftRows, InvMixColumns, and AddRoundKey) in reverse order. However, the order of InvSubBytes and InvShiftRows is indifferent. The decryption procedure thus can be rearranged as shown in Figure 1, where the InvRoundKey is obtained by applying InvMixColumns to the respective original RoundKey. Such a structural similarity in both the encryption and decryption procedures makes hardware implementation easier.

3. Proposed AES design

Xilinx hardware was selected for several reasons:

o It is the market.

o leader in re-programmable hardware.

o It is well supported by Celoxica The hardware was readily available

3.1. State vector

The state vector is 128 bits in size, and must be accessible byte-wise for substitution, column-wise for mix

columns, and diagonally for shift rows.For reasons discussed later, these operations require only four bytes at a time.

In line with the different types of RAM available on the Xilinx devices, there are three ways to store the state vector: in block RAM, distributed RAM or flip-flops. The block RAMs are separate from the CLBs and have their own dedicated routing. The lookup tables in the CLBs are used to implement distributed RAMs, and each CLB contains 4 flip-flops available for use as memory (slices). The access time for block RAMs is greater than flip-flop memory while the Block RAMs are limited in number and only have two ports, and are best suited to store large amounts of data referenced at relatively few points. Both ports on a single block RAM used with a data width of 16 bits (as shown in Figure 2) could meet the constraints for mix columns and substitution operations. Problems may occur when implementing the shift rows operations with this structure and additional "swap" memory may be required.

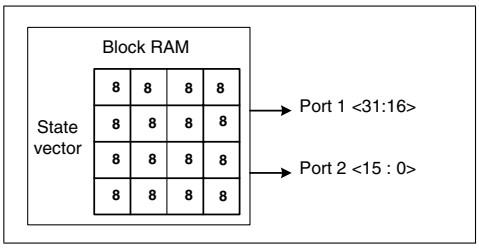

Figure 2. Block RAM implementation of state vector.

Regardless of the potential difficulty when implementing the ShiftRows operation, it was concluded that using only 128 bits was not the most efficient solution for a block RAM.

Flip-flops are the most expensive type of RAM, although they are the fastest and most flexible. Each slice can provide two flip-flops, and storing 128 bits would take 64 slices (32 CLBs). Distributed RAMs offer a reasonable compromise, but with a low access time, still cannot compete with the speed and flexibility of flip-flops. To achieve the 128 bits storage in distributed RAM while still meeting the accessibility constraints for Mixcolumns, SubByte and ShiftRows, we require 16 x 16 x 2 bits RAMs (see Figure 3). Each slice on the Spartan-II for example is capable of operating as a 16x2 bit RAM, therefore it will take 16 slices or 8 CLBs to store this structure. Storing just 128 bits in this manner is not particularly efficient, but it is still less demanding of hardware than using flip-flops. The final design used distributed RAM to store the state.

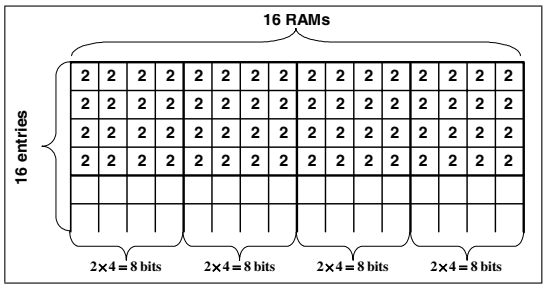

Figure 3. Distributed RAM implementation of state vector.

3.2. Key generation

Key generation can be implemented as either a separate unit or intrinsic to the overall design itself. The second option reduces logic depth but requires more hardware as the encryption and decryption can no longer share the same generator. Since its implementation was relatively small, this was the preferred option in our proposed design where there are only a few references to the key generator in the design.

3.3. Mix columns

All possible results of the multiplication operations and Multiplication boxes (M-box) can be stored in block RAMs. Given the size of the M-box, the only feasible alternative is to implement the required computational arithmetic operations in hardware. However, implementing this in hardware can be seen as a particularly expensive solution in terms of both performance and hardware cost, as it would require at least one multiplier - several perhaps for any practical design.

3.4. BlockRAM allocation

The number of block RAMs was the limiting factor for this design. It was critical that they be used efficiently. The key generator uses a block RAM to store the round constant vector. For full efficiency, four-port access to S-box is required and can be provided by two dual ported block RAMs.

In decryption the keys must be stored. Furthermore, they must be stored in block RAM as the key table is far too large to be feasibly stored in any other type of memory. In total, there are four block RAMs allocated to the key generator.

The device itself is capable of either operating as encryption or decryption operation, but not both at the same time. A single block RAM can be reconfigured to any of number of data widths. At 8 bits word length, a single RAM can

store 512 entries. Since substitution boxes and multiplication tables require only 256 entries, block RAMs have been split into halves (see Figure 4). Where the top half of each of the remaining block RAMs was allocated to encryption and the lower half to decryption, thus making available the full breadth of block RAMs to both encryption and decryption.

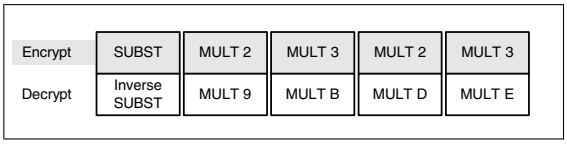

Figure 4. Split block RAM concept.

There are two block RAMs allocated to substitution boxes as shown in Figure 5. Such an allocation allows for four simultaneous accesses to the S-box (or inverse S-box), which is sufficient to complete the substitution lookups for a single 32 bit column of the state vector in any given clock cycle. To complete the substitution operation on the entire state vector would require four clock cycles.

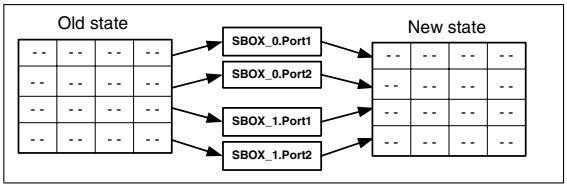

Figure 5. Substitution operation with four block RAM ports available.

For the MixColumns operation used in hardware, only MULT 2 (multiplication by 2) and MULT 3 (multiplication by 3) operations are required. This requires two M-boxes. To complete a MULT 2 operation on a single 32-bit column of the state vector requires four accesses to the M2-box. So that accessing both MULT 2 and MULT 3 operations would require eight accesses per column. In the decryption operation (for the inverse MixColumns operation), sixteen accesses are required per column - four for each of the MULT 9 (multiplication by 9), MULT B (multiplication by 11), MULT D (multiplication by 13) and MULT E (multiplication by 14) operations. An efficient design with an enhanced performance in terms of speed and area usage allows the decryption twice as many M-boxes, but requires only two accesses per M-box per clock cycle. Consequently, the calculation of the multiplication results for the decryption takes twice as long as the computation for encryption.

4. Experimental results

To assess the performance and the portability of our design, we have implemented our embedded architectures onto a number of FPGA devices: Spartan II , Virtex E, Virtex 2 and Virtex 4.

4.1. Column operations

There are insufficient blocks RAMs available within the FPGA device (Spartan II) used to execute the substitution, mix columns, or add round keys operation on the entire state vector in a single cycle. However, these operations instead operate column by column, and therefore require four iterations to complete. For example: executing the substitution operation on all 16 bytes in the state vector would require 16 lookups, to do this in one clock cycle would require 8 dual-ported block RAMs. Instead, two block RAMs are used, providing four ports. This allows the substitution operation to be completed in four clock cycles.

4.2. Key generation

Key generation can only occur in forward order. For the decryption operation, the keys must be generated and stored in a forward order, then made available to the decryption process in a reverse order. The process uses shift and XOR operations, and substitution and rcon (the round constant word array) lookup operations [2]. The key generator implementation summarised in Figure 6 was published in [12], and has been adopted in several designs.

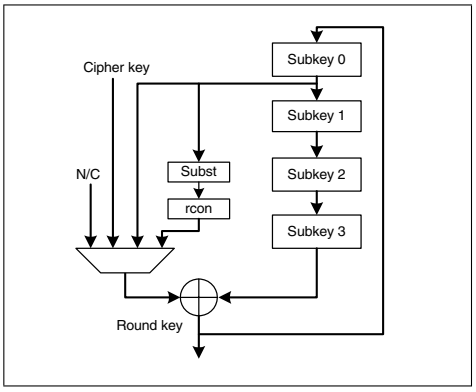

Figure 6. Example data path for key generator(32 bit data flows).

This process produces keys of 32 bits at a time. The multiplexer switches between the cipher key, the four byte key before the current key, and the same key post-substitution task. The key generator used in the final design was based on the above design but, in the case of the encryption

process, was modified when it was made a part of the AddRoundKey process.

4.3. Encryption round

The MixColumns operation takes place after substitution in the encryption round. Since the substitution and the M-box are pre-computed, a single set of pre-substituted M-box can be create at the expense of an additional memory required to store the substitution, MULT 2 and MULT 3 results prior to the XOR. The substitution and MixColumns can occur concurrently (see Figure 7), but in two steps: one for the block RAM lookups; and one to XOR the results. The cyclic shift operation in shift rows can be implemented concurrently, by saving the results of the XOR operation in the appropriate (shifted) places of the updated state vector.

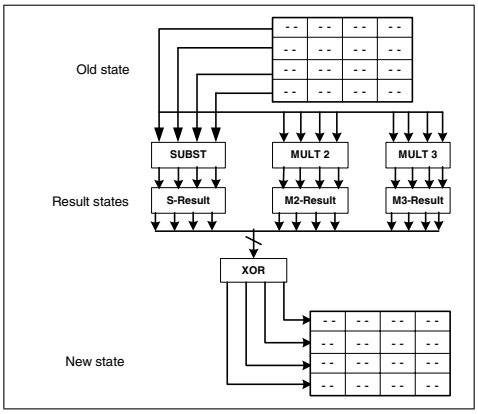

Figure 7. Concurrency of substitution and MixColumns in encryption.

The end result is that the ShiftRows operation, the generation of both the substitution table and multiplication tables can all take place in one step (see Figure 8). Furthermore, the generation of the next round key and the XOR with the current round key with the results can be performed in the second step.

4.4. Decryption round

It is worth noting that the concept used for the concurrency in encryption task is less effective for the decryption process. This is because the functions are fed in reverse order, thus pre-multiplying substituted values is not realistic.

Therefore, for the decryption, it is not possible to reuse the substituted values in the MixColumns operations. Furthermore, the inverse MixColumns XOR operations cannot be shared with the key addition or the substitution operations. Consequently, multiplication for the inverse MixColumns operation takes three steps: two for generating the

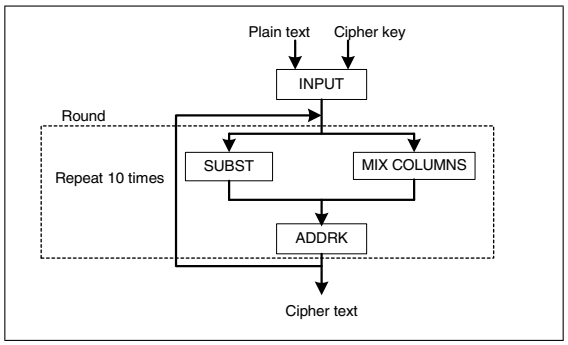

Figure 8. Encryption data path diagram.

multiplication tables, one more for the XOR to calculate the result of the inverse MixColumns. This is a weakness in the design.

The best that could be achieved for decryption given the limitations of this design (in particular the block RAM usage) is five steps: substitution, XOR with the generated round key, and a three-stage multiplication (see Figure 9).

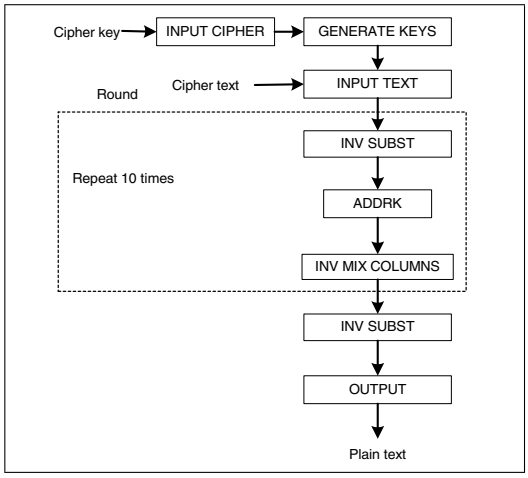

Figure 9. Decryption data path diagram.

4.5. Performance Evaluation

The device has 32 pins for the input text, 32 pins for the cipher key, 32 pins for the output text, two control pins (”start” and ”Enc/Dec”), and the standard power and clock pins. A basic guide to the device connections is shown in Figure 10.

To configure the device for either encryption or decryption, the mode pin must be set appropriately. ”Start” must be set high to trigger the process. If the pin is held high until the process has been completed, it will continue uninterrupted with the next available block. The maximum

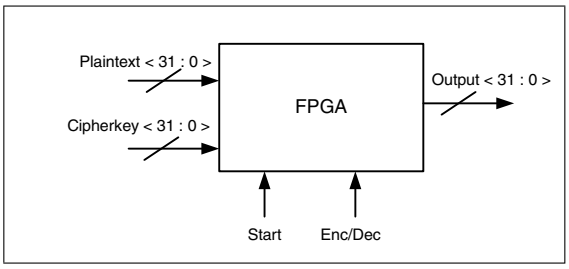

Figure 10. Device connections.

throughput is realised when this pin is held high permanently and new input blocks are made available as required.

For encryption, the "Enc/Dec" pin must be set high before "Start" is set high. The input text and cipher key are fed in concurrently.

For decryption, the "Enc/Dec" pin must be set low before "Start" is set high. The cipher key is required once the process is underway, but the cipher text is not required until the keys are generated (48 clock cycles after the start of the process).

At run-time, the encryption and decryption processes require 93 and 260 clock cycles, respectively, to complete a 128-bit input block. Table 1 provides a convenient summary and breakdown of the clock cycle requirements. Note that processes running within each round occur 10 times (recall there are 10 rounds in AES-128). In decryption, the MixColumns process takes 3 steps per column per round, and therefore takes 120 clock cycles to complete.

Table 1. Clock cycles breakdown.

Encryption		Decryption	
Stage	Cycles	Stage	Cycles
Input	5	Input cipher	4
Subst	40	Generate keytable	44
AddRoundKey	40	Input text	4
Final Subst	4	Subst	40
Output	4	AddRoundKey	40
		MixColumns	120
		Final Subst	4
		Output	4
Total	93	Total	260

5. Performance result and comparison

The implementation of the AES algorithm was made on many FPGA in order to show the portability of the algorithm. Handel-C codes are compiled and translated to EDIF (Electrical design Interchange Format) using Celoxica Design Kit (DK4) [13]. EDIF data is mapped to FPGA chips using a Xilinx synthesis tools (ISE 7.1i) [14].

In this section, we compare the results obtained using Handel-C to design our AES architecture against those obtained using other methodologies.

Table 2 summarizes the comparison. From the table, it can be seen that our design is the only one to use Handel-C among the group under study. Other designs used different methodologies, e.g., C^{++}, SystemC, VHDL.

Table 2 also shows that our design outperforms most of the other designs in speed. The only exception is Zhang H and al [18] and McLoone M and al [17]. However, this design requires larger area than ours. The reason partially lies in the fact that different goals were pursued: instead of striving for a fast solution suitable for specific FPD structure we tried to fulfil the requirements of the given target application by direct translation of C^{++} code to Handel-C design of the FPGA supported by the available development tools. The penalty of such an approach is that lower execution times are obtained.

When we compare the works [15] and [16] with our approach, one can note the advantage of use Handel-C in relation to C^{++} and SytemC on speed and area. Handel-C and SystemC languages can be used to design a target project from a high-level specification. However, SystemC has a limitation that it needs to be translated to register transfer level (RTL) to be synthetised, whereas handel-C has the capability to be mapped directly to hardware.

It is to be noted that our design requires more area than the study in [19] but has a much improved speed of operation.

These results have been achieved mainly because Handel-C is fully synthesizable; it supports parallelism and has the capability to be mapped directly to hardware. It also has the advantages of requiring less effort while providing better portability among different field programmable devices (Spartan, Virtex E, Virtex 2, Virtex 4,etc).

6. Conclusion

In the work presented in this paper, we have applied an emerging design methodology to design a reconfigurable AES architecture. This methodology uses the Handel-C language to design a target system and directly map it onto FPGA platforms. This methodology reduces the risk of errors of the traditional design trend, which prototypes a design in high level languages and then translates it into HDLs. In addition, this methodology increases the flexibility of a design with enhanced performance. Using this methodology, we have increased the AES unit performance through application of parallelism and pipelining. We have shown that higher performances could be achieved by using the Handel-C based methodology when compared to other works. Also, the time required to design, implement, and test the designed unit using this methodology is reasonably

Table 2. Comparison with other design methodologies.

Designs		[15]	[16]	[17]	[18]	[19]	Our design			
							A	B	C	D
Methodology		C^{++}	System C	VHDL	VHDL	VHDL	Handel-C	Handel-C	Handel-C	Handel-C
FPGA vendor		-	Xilinx	Xilinx	Xilinx	Xilinx	Xilinx	Xilinx	Xilinx	Xilinx
FPGA chip		-	XCV2000E	XCV812e	XCV1000	XC2S15	XC2000E	XCV1000	XC2S100	XC4VLX160
No. of Blocks RAMs used		-	104	244	No	2	9	9	9	10
Area (slices)	Encryption	-	8009	2000	11014	122	424	437	423	437
	Decryption	-	-	-	-	-	513	528	514	526
	Both	-	-	-	-	-	627	645	631	641
Throughput (Mbps)	Encryption	70.5	0.074	12020	16032	2.18	565	360	455	1716
	Decryption	-	0.071				218	145	189	624
	Both	-					160	116	140	460

low as compared to the time required using other design approaches.

References

[1] W. Stallings. Cryptography and Network Security, Prentice Hall, 2003.

[2] FIPS PUB 197, Advanced Encryption Standard (AES), NIST, U.S. Department of Commerce,November 2001. (http://csrc.nist.gov/publications/fips/fips197/fips-97.pdf).

[3] Tessier, R., and Burleson, W.:'Reconfigurable computing for digital signal processing: a survey', J.VLSI Signal Process.,2001,28,(1-2), pp.7-27

[4] Celoxica Limited: 'Handel-C language overview'. Product Brief. August 2002. http://www.celoxica.com/techlib/files/CEL-W0307171KDD-47.pdf

[5] Holland, B., Vacas, M., Aggarwal, V., DeVille, R., Troxel, I., and George, A.D.: 'Survey of C-based application mapping tools for reconfigurable computing'. Proc. 8th Annual Int. Conf. on Military and Aerospace Programmable Logic Devices (MAPLD 2005), Washington, DC, USA, September 2005

[6] Celoxica Limited: 'Handel-C language reference manual'. 2003.[Online].Availableat:http://www.celoxica.com/techlib/files/CEL-W030811132Q-60.pdf

[7] Aubury, M., Page, I., Randall, G., Saul, J., and Watts, R.: 'Handel-C language reference guide'. August 1996. (http://www.inf.pucrs.br/ moraes/topicos/hdls/HANDELC/HANDELC.PDF)

[8] Open SystemC Initiative (OSCI): "Draft standard SystemC language reference manual". April 2005. [Online]. Available at: http://www.systemc.org

[9] Perez, D.G., Mouchard, G., and Temam, O.: "A new optimized implementation of the SystemC engine using acyclic scheduling". Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE 2004), Washington, DC, USA, February 2004, pp. 552-

[10] J. Daeman and V. Rijmen, "The Design of Rijndael: AES - The Advanced Encryption StandardJoan Daeman and Vincent Rijmen". Springer Verlag, 2002.

[11] "http://www.esat.kuleuven.ac.be/ rijmen/rijndael/."

[12] Grembowski, T, "FPGA 3-in-1 Hardware Implementation of Rijndael Cipher". http://ece.gmu.edu/courses/ECE636/project/reports/TGrembowski.pdf, 4 May 2001.

[13] Celoxica Limited: "Software product description for DK version 4.0".December 2005.

[14] Xilinx.(2005) Xilinx ISE 7 software manuals and help. (http://toolbox.xilinx.com/docsan/xilinx7/books/manuals.pdf)

[15] B. Gladman, "AES Second Round Implementation Experience, AES Round 2 public comment". May 15, 2000, available at http://www.nist.gov/aes

[16] Jae-Gon Lee, Woong Hwangbo, Seonpil Kim, Chong-Min Kyung, "Top-down Implementation of Pipelined AES Cipher and its Verification with FGPA-based Simulation Accelerator" in Proc. The International Conference on ASIC (ASICON 2005), Shanghai, China, Oct. 2005, pp.140-143.

[17] McLoone, M., McCanny, J.V. "Rijndael FPGA Implementations Utilizing Look-Up Tables" Journal of VLSI Signal Processing Systems, KAP, vol. 34-3, pp 261-275, July 2001.

[18] X. Zhang and K. K. Parhi, "High-speed VLSI architectures for the AES algorithm," IEEE Trans. Very Large Scale Integation (VLSI) Syst., vol. 12,no. 9, pp. 957-967, Sep. 2004.

[19] Tim Good and Mohammed Benaissa, "Very Small FPGA Application-Specific Instruction Processor for AES," in IEEE transactions on circuits and systems",regular papers, Vol.53, No.7, july 2006.

0-7695-2866-X/07 $25.00