Cryptographic Functions on the SideWorks architecture

Nuno Pissarra
nuno.pissarra@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2015

Abstract

This work presents the implementation of multiple cryptographic algorithms, the SHA family, AES and CLEFIA using the Coreworks processing framework. The FireWorks processor of this framework will be used to control the hardware accelerator designated as SideWorks. The presented work explores the use of the SideWorks platform to adequately schedule and accelerate the computation of the most common cryptography algorithms (excluding asymmetrical algorithms). The proposed approach considers merging the needed processing structures, towards more compact and efficient cryptography implementations.

Keywords:SHA, AES, CLEFIA, CoreWorks, SideWorks, FPGA

1. Introduction

1.1. Motivation

Today modern world of e-mail, Internet banking, online shopping and other sensitive digital communications, cryptography has become a vital tool to ensure the connection and interaction in society. In order to provide confidentiality and authentication among other security services, cryptography primitives need to be employed when sending sensitive information through these channels. Cryptography algorithms have been in use for a long time, but the growing processing capability of digital equipment and the growing bandwidth in digital communication impose the need for more dedicated solutions.

1.2. Development environment

Coreworks developed a computing technology which minimizes the problems outlined above as it speedsup the development of high-performance, small area and low power reconfigurable processors. The Coreworkstechnology named SideWorks highlights that reconfigurable processors built with this technology are primarily targeted to work as a dedicated high-performance offload engines. SideWorks is a general architecture template for runtime reconfigurable processors, using pre-designedand pre-verified programmable Functional Units (FUs) and embedded memories interconnected by programmablepartial crossbars.

With this technology, application-specific architectures can be automaticallygenerated, programmed and simulated by proprietary tools. The Coreworks technology combines a proprietaryconventional processor (FireWorks) with a programmable hardware block (SideWorks), to form a totally programmablesolution which achieves the performance of custom hardware and the flexibility of software.

1.3. Objectives

When facing the challenge to implement cryptographic algorithms in a platform such as SideWorks,the developer must identify the necessary building blocks to achieve the most efficient and compact implementation. As such, the goal of this work is not only to implement cryptographic algorithms in SideWorks but to provided SideWorks withessential building blocks to do so in the most effective way. That there are no current implement patrons of this kindin SideWorks is the motivation behind this project and could represent a new field in which SideWorks can thrive.

1.4. Document Structure

The document is organized as follows: Section 2 and 3 presents the general description of the target algorithms using reference implementations provided by the authors; Section 4 describes the development environment and respective characteristics of the SideWorks platform. Section 5 and 6 introduces the proposed structures, presenting the support procedures and mechanisms devised during the course of the work. Section 7: The simulation results of the proposed structures are provided as well as an overview of how to the results were obtained. Section 8: Presents the conclusions and future work that can be developed from the work developed in this thesis.

2. Hash Functions

Hash functions are a key primitive in modern cryptography. Cryptography hash functions are unidirectional functions that map binary strings, of arbitrary length, to fixed length values, called the hash-values or the digest value. Hash functions are used in digital signature scheme, message padding of public key encryption scheme and message authentication codes. The hash functions targeted in this workconsists ofthe three versions of the Secure Hash Algorithm (SHA) family published by NIST.

2.1. SHA-1

The SHA-1 produces a single output 160-bit message digest (the output hash value) from an input message. The input message is composed of multiple blocks. The input block, of 512 bits, is split into 80×32-bit words, denoted as W_t, one 32-bit word for each computational round of the SHA-1 algorithm. Each round comprises additions and logical operations, such as bitwise logical operations (F_t) and bitwise rotations to the left as depicted in Figure 1. The calculation of F_t depends on the round

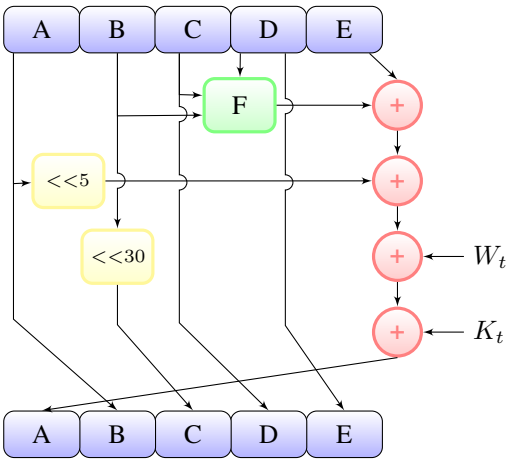

Figure 1: Schematic of the SHA-2 algorithm.

t being executed, as well as the value of the constant K_t. The SHA-1 80 rounds are divided into four groups of 20 rounds, each with different values for and the applied logical functions (F_t) present in equation 1.

$$\begin{array}{ll} (B \wedge C) \vee (\bar{B} \wedge D) & 0 \leq t \leq 19 \\ B \oplus C \oplus D & 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & 40 \leq t \leq 59 \\ B \oplus C \oplus D & 60 \leq t \leq 79 \end{array} \quad (1)$$

The initial values of the A to E variables in the beginning of each data block calculation correspond to the value of the current 160-bit hash value. After the 80 rounds have been computed, the A to E 32-bit values are added to the current state. The Initialization Vector (IV) for the first block is a predefined constant value. The output value is the final state, after all the data blocks have been computed.

2.2. SHA-2

The hash functions of the SHA2 family differ most significantly in the number of security bits that are provided for the hashed input message. The SHA-2 standard supersedes the existing SHA-1, adding four new hash functions, SHA224, SHA256, SHA384 and SHA512, for computing a condensed representation. The produced message digest ranges in length from 224 to 512-bits, depending on the selected hash function. The SHA-2 algorithm is very similar in structure to SHA-1 nevertheless it uses eight, rather than five, 32-bit words (64-bit in SHA384 and SHA512), and operates in the same manner of SHA-1. The message to be hashed is first: (1) padded with its length in such a way that the result is a multiple of 512 bits long, and then; (2) parsed into 512-bit message blocks $M_{(1)}$, $M_{(2)}$,...,$M_{(N)}$. The compression function is depicted in Figure 2. The message blocks are pro-

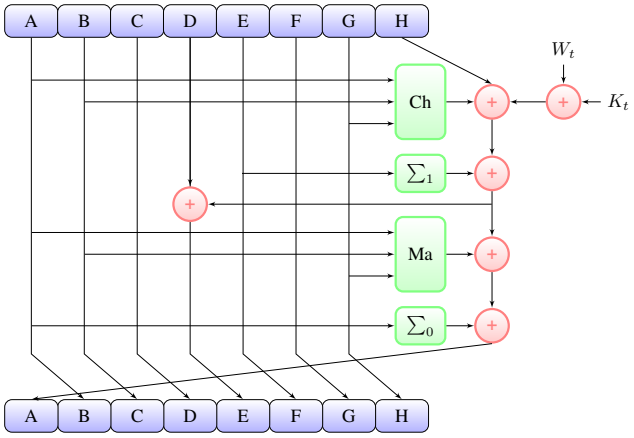

Figure 2: Schematic of the SHA-2 algorithm.

cessed one at a time, beginning with a fixed initial hash value $H_{(0)}$, sequentially computing $H_{(i)}$= $H_{i-1} + C_{M_i}(H_{i-1})$ where C is the SHA-2 compression function and + means word-wise mod 2^{32} addition. $H_{(N)}$is the hash of M.

2.3. SHA-3

SHA-3[1] is a family of sponge functions, where the underlying function is a permutation chosen in a set of seven permutations, where b $\in$ {25, 50, 100, 200, 400, 800, 1600} is the width of the permutation. The state is organized as an array of 5 x 5 lanes, each of lengths $w \in$ {1, 2, 4, 8, 16, 32, 64} ($b = 25w$). The choice of operations is limited to bitwise XOR, AND, NOT and rotations. The number of rounds n_r depend on the permutation width, and is given by $n_r = 12 + 2l$, where $2^l = w$. This gives 24 rounds for the nominal version of SHA-3 which operates on a 1600-bit state. The compres-

sion function of SHA-3 consists of 5 steps for each round , Theta (θ), Rho (ρ), Pi (π), Chi(χ) and Iota (ι) as shown in eq. 2 to 6.

θ step: $0 \leq x, y \leq 4$

$$\begin{aligned} C[x] &= A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] \\ D[x] &= C[x-1] \oplus rot(C[x+1],1) \\ A[x,y] &= A[x,y] \oplus D[x] \end{aligned} \tag{2}$$

ρ and π steps: $0 \leq x, y \leq 4$

$$B[y, 2*x+3*y] = rot(A[x,y], r[x,y]) \tag{3}$$

χ step: $0 \leq x, y \leq 4$

$$A[x,y] = B[x,y] \oplus ((notB[x+1,y])andB[x+2,y]) \tag{4}$$

ι step: $0 \leq x, y \leq 4$

$$A[0,0] = A[0,0] \oplus RC \tag{5}$$

Where A denotes the complete permutation state array, and *A[x,y]* denotes a particular lane in that state. *B[x,y]*, *C[x]*, *D[x]* are intermediate variables. The constants *r[x,y]* are the rotation offsets, while *RC[i]* are the round constants. The *rot(W,r)* is a bitwise cyclic shift operation, moving bit at position i into position i+r.

3. Ciphering algorithms

Most of the currently used security and authentication protocols are often supported by two main types of cryptographic functions, namely symmetric and asymmetric. However, in embedded devices, in particular, the symmetric key algorithms are a core component used to cipher and validate the transited data. Two examples of symmetric key algorithms are AES and CLEFIA, where the first is the most widely used symmetric-key cipher today, and CLEFIA represents an alternative to AES developed by SONY, also known to be a lightweight algorithm. Both AES and CLEFIA are symmetrical block ciphers sharing some ciphering techniques such as diffusion matrices, permutations, and byte substitution. The major differences reside on the fact that CLEFIA is based on a Feistel Structure while the AES algorithm is based is based on the Substitution Permutation Network structure.

3.1. AES

The basic operations used in the AES algorithm can all be described very easily in terms of operations over the finite field Galois field (GF)(2^8). The round transformation modifies the 128-bit state. The initial state is the input plaintext and the final state after the round transformations is the output ciphertext. The state is organized as a 4×4 square matrix of bytes

The round transformation scrambles the bytes of the state either individually, row-wise, or column-wise by applying the functions SubBytes, ShiftRows, MixColumns, and AddRoundKey sequentially. An initial AddRoundKey operation precedes the first round. The last round differs slightly from the others, the MixColumns operation is omitted. The functions of the round transformation are linear and non-linear operations that are reversible to allow decryption using their inverses. Every function affects all bytes of the state. The function SubBytes is the only non-linear function in AES. It substitutes all bytes of the state using table lookup. The content of the table can be computed by a finite field inversion followed by an affine transformation in the binary extension field GF(2^8). The resulting lookup table is often called an S-Box. The same S-Box is used for all 16 bytes of the state.

The ShiftRows function is a simple operation. It rotates the rows of the state by an offset. The offset equals the row index, the first row is not shifted at all, and the last row is shifted 3 bytes to the left. The MixColumns function accesses the state column-wise, working on each column in the same way. It interprets a column as a polynomial over GF(2^8), with degree < 4. The state bytes are the coefficients of the polynomial. The output column corresponds to the polynomial obtained from the multiplication by a constant polynomial and reducing the result modulo $x^4 + 1$.

The AddRoundKey function is a 128-bit XOR operation that adds a round key to the state. A new round key is derived for every iteration from the previous round key. The initial round key is equal to the original secret key. The computation of the round keys is based on the SubBytes function and uses additionally some simple byte level operations like XOR.

Decryption computes the original plaintext of an encrypted ciphertext. During the decryption, the AES algorithm reverses encryption by executing inverse round transformations in reverse order. The round transformation of decryption uses the functions AddRoundKey, InvMixColumns, InvShiftRows, and InvSubBytes in this order. AddRoundKey is its own inverse function because the XOR function is its own inverse. The round keys have to be computed in reverse order. InvMixColumns needs a different constant polynomial than MixColumns does. InvShiftRows rotates to the right instead of to the left. InvSubBytes reverses the S-Box lookup table by an inverse affine transformation followed by the same inversion over GF(2^8) which was used for encryption.

3.2. CLEFIA

The CLEFIA algorithm is a 128-bit block cipher, based on a 4-branched Feistel network, accepting key lengths of 128, 192 and 256 bits, processed over 18, 22, or 26 rounds. The 128 bits (16 bytes) of plain text are arranged into an array of four 32-bit words, processed over N round sets of instructions. The first step of the encryption process is to

XOR the second and fourth words of the plain text with the first and second 32 bits of the original key, performing a key whitening procedure. After this operation, the rounds are executed.

In each round, the four input words are processed, where copies of the first and third input words go through a 32-bit output non-linear function, F_0 and F_1 respectively, which consist of Round Key (RK) addition, four nonlinear 8 bit S-boxes, and a diffusion matrix. After each one, the result is XORed with the second and fourth words. The resulting four words are then swapped by left-round shifting them. After all rounds are computed, the final output values are obtained by XORing, one last time, the second and fourth final words with the last two Whitening Keys, instead of the last word swap.

4. SideWorks

SideWorks core has a fully customizable architecture targeted for reconfigurable digital signal processing applications. The SideWorks design is based on a coarse-grain reconfigurable array and is optimized for the efficient execution of algorithms comprise of nested loops containing complex logic and arithmetic expressions. These computationally expensive algorithms are common in a vast number of application domains. Additionally, the post-silicon reconfigurable feature allows a reduction of silicon area by combining multiple fixed-function hardwired blocks in fewer SideWorks instances.

SideWorks key features include pre-silicon configurability of nested loop depth, number of input and output ports, type and number of arithmetic FUs, datapath width, datapath routing, and address generation. In addition, the core supports runtime partial reconfiguration using a memory mapped configuration register file.

This Digital Signal Processor (DSP) technology targets cost and power sensitive applications, such as multimedia and communications. SideWorks enables the creation of DSP cores that are both configurable before fabrication and reconfigurable. The data transfers and some aspects of the execution unit function are programmable at runtime. SideWorks does not run as a stand-alone processor instead it is combined with a general-purpose host processor that manages program flow and data I/O. Therefore, Coreworks also supplies FireWorks, a compact, 32-bit RISC processor core. Designed to run computation intensive applications in a multiprocessor environment the Coreworks Processing Engine (CWPE) is an Intellectual Property (IP) a core system developed by Coreworks.

CWPE computational power comes essentially from the use of Coreworks reconfigurable accelerator technology – SideWorks blocks.

A high-level view of the SideWorks architecture is depicted in Figure 3. The architecture consists of arrays of FU, Memory Units (MUs) and Address Generation Units (AGUs), which can be flexibly and dynamically interconnected according to the information in a configuration register file *(CONFIG)*. A second register file *(CONFIG NXT)* holding the next SideWorks configuration is included for fast reconfiguration of the engine. The data

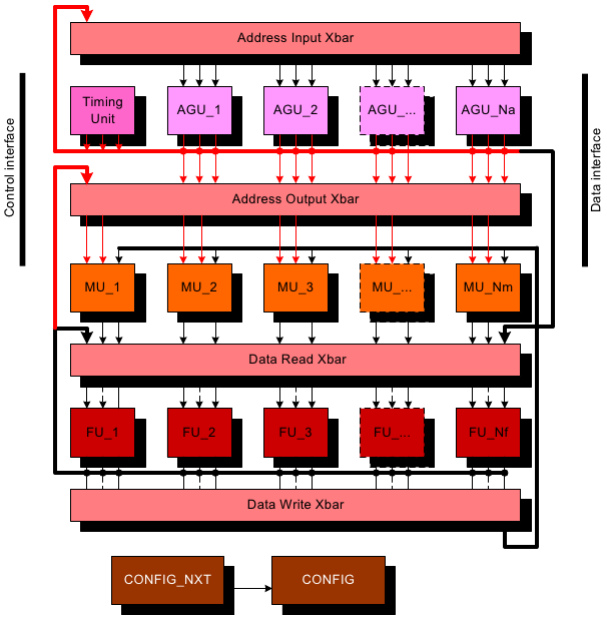

Figure 3: Top level view of the SideWorks architecture template[3].

defining the configuration of the programmable FU, crossbars, and address generators can be found in the configuration register file. It also stores some constants used in the computations. The SideWorks configuration register file can be accessed by the Direct Memory Access (DMA) while the engine is running also hiding or mitigating the reconfiguration time. SideWorks tasks are modeled as sequences of nested loop groups in C, using a proprietary function library called SideC. The values in the configuration registers create a temporary hardware datapath for executing a nested loop group[3].

5. Proposed Hash solutions

As mention in Section 2 all the target hash functions require only a small number of operations such as *add*, *and*, *xor*, *not* and *rotation*, which are available in SideWorks, at a minimal cost of one cycle per operation, which is far from optimize. With the exception of 64 bits operations such as *add* and *rotation* required by SHA512 and SHA-3 respectively. In the SHA-3 case the chosen implementation solves this problem by unrolling 4 rounds and using factor 2 interleaving technique to map the 64-bit lanes to 32-bit words and representing the 1600 bits state as 50 words of 32 bits in order to save memory. Leav-

ing the 64 bits adder required by SHA512 problem unsolved, to this end a FU named $ADDX2$ was created.

This is one of the reasons why the first step in the implementation is to evaluate the availability of the logic functions and all of the necessary FU in SideWorks. In order to either adapt the structure to fit the platform limitation or creating new FUs to improve the overall speed and area requirements. During the design and developing stages one of the main objectives when adding a new FU is to try to make it the most generic as possible in order to enable the reuse of the same FU among multiple datapaths seeking to optimize the solution.

A good example is provided by the SHA-1 implementation where all the necessary logic functions required by the algorithm were available as FUs, yet analysis of the critical datapath when using the pre-existing FUs, all of three branchs of the SHA-1 F_t function (equation 1) would require eight cycles per round.

Breaking down the SHA-1 F_t branchs, f_1 employs two AND operation, one NOT and one OR, this requires three cycles; f_2 requires only two $XORs$, to obtain the result of $XORing$ three words, which has a cost of two cycles; In the other hand F_3 employs two OR plus three AND operations at the cost of three cycles per round.

The F function could be simply implemented as a custom FU, but that choice would be against the main idea of the reusability of the FUs. With this in mind we spilt the F function in two distinct FUs in order to improve the reuse possibilities. The F_1 and F_3 operations are available in FU named $ANDX$ and the F_2 in $XORX$. The FU named XORX which represents a three input XOR with the particularity of supporting the rotations required by SHA-1 and SHA256.

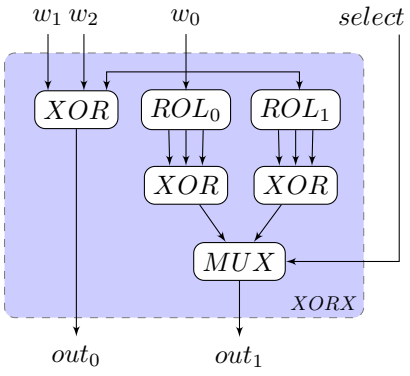

Figure 4: XORX FU description

Taking in to account the use of the newly proposed FUs, that will execute the F_t function on a single cycle, it is necessary to create a multiple input adder with a select flag in order to correctly choose the output from the F_t function FUs. A FU called $ADDX$ was developed with seven 32-bits word input and two one bit $selectflag$ input, performing a five word adder with a selection of which of the last three words is used for the operation, this provides the ability to select the F branch. A breakdown of the operation is depicted in Figure 5.

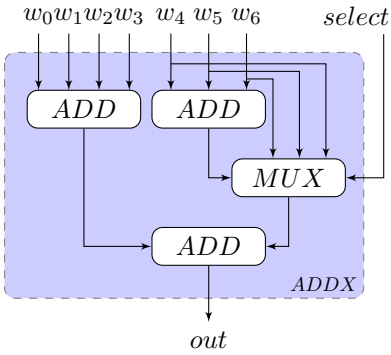

Figure 5: ADDX FU description

The final structure for the SHA-1 algorithm includes one $ADDX$, one $ANDX$, two ROL and one $XORX$, all new FUs addes to the SideWorks platform in order to provided a efficient implementation. Using the FUs outline the implementation of the SHA-1 algorithm in the SideWorks platform is possible, requiring only four cycles per round against the eight necessary for the implementation using only the available FUs. Considering eighty which is the number of rounds of the SHA-1 algorithm multiplied by four (number of cycles per round) results in three hundred and twenty cycles required to hash a value using this implementation SHA-1 in SideWorks.

The critical datapath for the SHA-1 structure is illustrated in Figure 6 were the elements presented with a red background are used to represent memory blocks that require two clock cycles (for the read and write operation), or as an alternative, only one clock cycle for each of the operations when using registers. The blue elements represent FUs, always single cycle latency.

Following the same approach a total the 4 datapaths were created to provide a working versions of SHA-1, SHA256, SHA512 and SHA-3512. Promoting the reusability of FUs among datapaths, which is present in all designed structures, such as the SHA-1 and SHA-2 were by exploiting the similarities between the algorithm a design the used FUs accordingly, the FUs used in SHA-1 are reused in SHA-2. Meaning if the SHA512 datapath is used little overhead is required to implement the SHA-1 and SHA256 respective datapaths due to the shared number of FUs.

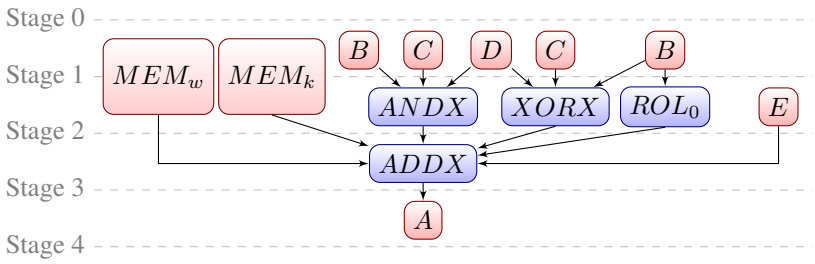

Figure 6: SHA-1 Final critical datapath.

In the SHA3 algorithm although it requires the same operations as the other algorithms the implementation itself is not so trivial due to the structure of the algorithm. As such the proposed structure uses specificity designed FUs as well as FUs from all the other structures. This implementation is based on a factor-2 interleaving technique created by the authors that allows efficient implementations on systems with small word sizes [1].

6. Proposed Ciphering solutions

Likewise, the AES [7] and CLEFIA [5] structures share most of the same FUs it is important to note that for these algorithms a LookUp Table (LUT) coupled with T-Box methodology based was chosen, resulting in fully rolled implementation of AES [2, 4] and CLEFIA[6].

In the case of AES, using the implementation described by *Chaves et al.* in [2] as a starting point, three new FUs were design, $CXOR$, $SXOR$ and $FXOR$. Figure 7 presents the inner works of this $SXOR$ FU where the red values represent the switch in the index for the decryption mode.

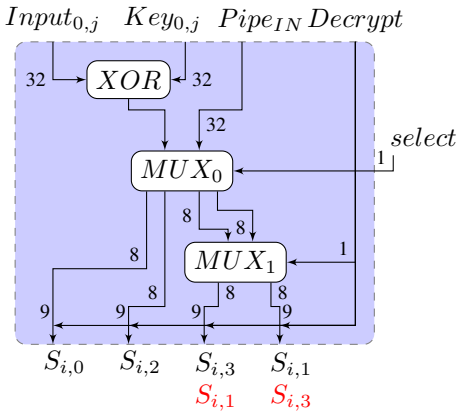

Figure 7: XOR Select description

Each the new FUs corresponding to a different stage of the algorithm. From the beginning, the $SXOR$ was conceived to select and route, the initial and feedback inputs for the rounds. Accepting as input the Round Key for the first round, the plaintext input and the result from the previews rounds as well as a mode select in order to determine when to use decryption. The output of which is the directly connected to the address of the T-Box memory, as such the value obtained from the T-Box memory needs to be XORed to complete the round, for that propose the $CXOR$ was created. The result of the round is computed by XORing the output of the T-Box. The last round of the AES computation has the particularity of not applying the $MixColumn$ operation, as illustrated in the in Figure 8. This can be computed with the memory structure which only performs the $SBox$ operation. Also, the output value is directly added to the key since no polynomial addition has to be performed.

In Figure 8 a part of the AES Electronic Code Book (ECB) core is shown, which processes 32 bits block of the 128 bits *state* of the AES algorithm, as such the full structure is composed of a total four structures like the one represented, each one processing 32 bits. Ever 128 bits block requires four clock cycles in order to be processed. Were the elements with a red background are used to illustrate memory block which requires two clock cycles both for the read or write operation, and the blue elements represent FUs that have a single cycle latency.

A second version of this structure is proposed, the main difference is the added ability to use Cipher Block Chaining (CBC) mode and also compatible with ECB mode. To enable this feature, the second version of $SXOR$ a $FXOR$ was created, supporting the same functions as the first version but adding one more input *word* and a *select* flag used to determine when the CBC mode is to be employed, and the new *word* input to accept the respective IV.

The proposed implementation uses a fully rolled version of the AES core capable of encrypting and decrypting data blocks in both ECB and CBC modes for all key sizes, i.e., for 10, 12 or 14 rounds. The operation modes, as well as the IV for the CBC mode, are passed as additional parameters.

Furthermore to optimize the implementation Block Memory (BRAM) is used, available through

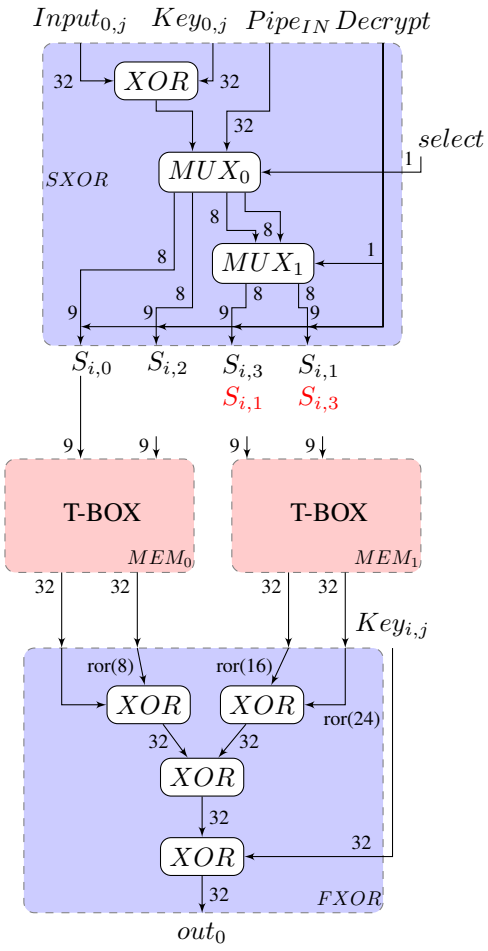

Figure 8: AES ECB Core Partial Round Example

the SideWorks framework, which enables the use of dual port RAM cutting in half the memory requirements, ideal option in order to save the logic resources.

$$Throughput_{AES128} = \frac{80(\text{MHz}) \times 128(\text{bits})}{40} \quad (6)$$

7. Evaluation

In this Section, experimental results for the proposed structures are presented and compared.

The obtained results regarding the FUs was obtained using the Xilinx ISE WebPACK Design Suite (v12.4) with the FU designs described using VHDL. For all the FU, a respective test bench was created using known values for the target algorithms in order to validate the FU correct operation. The values presented for the proposed designs were obtained after Place and Route processing with software default parameters, namely Synthesis Normal Speed Optimization Effort, and High Optimization Effort in Mapping and Place and Route, with no extra effort. All the obtained results for the total number of clock cycles required by each datapath are provided by the SideWorks simulator present in the Coreworks developer tools.

7.1. Hardware Tests

All the hardware tests were realized using a Spartan3 xc3s5000 gently provided by Coreworks, on this FPGA the clock obtained in the SideWorks platform was 80 MHz, this is the value used as reference. The Spartan3 FPGA was a shared resource among the developers working in Coreworks with product debug and testing taking priority.

Nevertheless with the help and patience of the Coreworks engineers it was possible to test the FU that compromise the SHA-1 structure as well as the AES-ECB implementation. This includes a test bench for each of the FU present in the developed structures, were by using multiple known values obtain for the software implementation of the algorithms it was possible to verify the correct operation of each FU. This phase was also very important to assess some of the limitation of the platform, in particular, the latency associated to each FU. Initial tests included more complex FU, as a consequence of the complexity the FU presented a high Latency values which represented a problem as this higher Latency values would translate in the FU requiring two o more clock cycles to complete the target operation. As such this line of developed was discontinued and the FU described in this work all present low Latency considering the reference value for the SideWorks implementation. From all the developed FU present in this work the one with the higher Latency value is $ADDX$ with $8.5ns$, supporting an operating frequency up to 117 MHz, far below the reference value for the SideWorks platform. Confirming that $ADDX$ FU requires only one clock cycle to complete its operation properly.

During this phase many bugs were found and corrected although the greater majority of them were a byproduct of the ongoing rework of the framework taking place at the time of this testing. All the corresponding bugs checked by a Coreworks engineers but at that time considered low priority since they did not affect the main products. Working around some memory issues present at the time, it was possible to validate the SHA-1 and AES ECB implementation. Due to the lack of time and the limited resources this was the full extent of the hardware tests, all the subsequent results are provided by SideWorks simulator present in the Coreworks developer tools

7.2. Functional Unit Design

During the design and developing stages one of the main objectives when adding a new FU was to try the most generic approach possible in order to enable the reuse of the same FU among multiple datapaths seeking to optimize the solution the best pos-

	SHA-1	SHA256	SHA512	SHA-3	AES ECB	AES CBC	CLEFIA
Block Size (bits)	160	256	512	1600	128	128	128
Cycles per round	4	5	5	5	4	4	4
Total Nbr Cycles	320	320	400	120	40	40	72
Throughput (Mbps)	40	64	102	1067	256	256	142
Slices	417	674	1348	32320	416	692	192

Table 1: Proposed structures performance summary

sible way. Even considering all the merits of this approach it can be extremely counterproductive.

A good example of this is the $CXOR$ FU which receives as one of its inputs an integer to be used as a $factor$ on a rotation executed by this specific FU. The $factor$ can take a great number of values, but a careful analysis of the algorithms that require the FU, it becomes apparent that only two values are ever used as input $factor$, they are 1 and 8. Taking this into account the FU can be largely improved, by reducing the number of values considered for this variable or removing it.

To fully leverage the capability of SideWorks it is extremely important to know how to select and when to reuse FU. This is why multiple approaches most be considered when designing FU in order to optimize the final results. Well define requirements from the beginning of a project will greatly reduce the chance of mistakes like this to manifest in later stages of the project. Although due to the versatility of the platform situations like these can be corrected quickly.

7.3. Proposed Structures performance

Table 1 presents a performance overview of the proposed structures, in order to derive the values in the table, it is necessary to take into account the reference values, such as 80 MHz for the maximum clock speed when considering the Coreworks platform running in a Spartan3 xc3s5000 FPGA.

These are the reference values used to calculate the throughput as exemplified in equation 6, where 40 is the number of cycles required per block also shown in the table, resulting in $256Mbps$.

Just as it is important to refer that the area values only account for the total number of slices required by the FU used by each of the proposed structures and not the total amount considering the SideWorks platform. Due to the fact that to synthesize the design and perform the Post-place&Route procedures for the SideWorks platform plus the developed structures requires access to SideWorks source files only available to Coreworks engineers, this was only an option during the hardware testing phase.

8. Conclusions

The development of this work provided the CoreWorks framework with some of the building blocks required by many cryptographic algorithms. Coupled with the functional implementation of multiple algorithms, this is a good starting point for the developed of a commercially viable cryptographic core on top of CoreWorks framework. The CoreWorks provides a fast paced development environment with a lot of potential, and the unique ability to facilitate the creation of tailor-made solution in order to suit the clients needs. Improvements can easily be made by choosing a smaller subset of algorithms especially if those share the same structure or part thereof, making it possible to take full advantage of the benefits provided by this platform. Reducing the necessary area and via the reuse of FU and reducing the time required to execute each algorithm.

References

[1] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011.

[2] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa. Reconfigurable memory based AES co-processor. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[3] Coreworks. *SideWorks Reference Book*. Coreworks S.A., 2010. Revision v1r2.

[4] T. Good and M. Benaissa. AES on FPGA from the Fastest to the Smallest. In J. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 427–440. Springer Berlin Heidelberg, 2005.

[5] Security techniques - Lightweight cryptography - Part 2: Block ciphers, 2000.

[6] T. Kryjak and M. Gorgon. Pipeline implementation of the 128-bit block cipher clefia in fpga. In M. Danek, J. Kadlec, and B. E. Nelson, editors, *FPL*, pages 373–378. IEEE, 2009.

[7] N. I. of Standards and Technology. Advanced Encryption Standard. *NIST FIPS PUB 197*, 2001.