Polymorphic AES Encryption Implementation

Ricardo Chaves, Leonel Sousa
Instituto Superior Técnico / INESC-ID
Portugal, Lisbon
Email: ricardo.chaves@inesc-id.pt

Georgi Kuzmanov, Stamatis Vassiliadis
Computer Engineering Lab, Delft University of Technology
Delft,The Netherlands

***Abstract*— This paper presents a hybrid hardware-software implementation of the AES encryption algorithm on the MOLEN polymorphic processor [1]. In order to combine the advantages of both the software and the hardware implementations, the application code has been divided into two computational approaches, software and hardware. Only the main ciphering function, the more computational demanding component, has been implemented in hardware in a Virtex II pro-20. The AES encryption core occupies about 20% of the available slices and utilizes 12 BRAMs. The proposed design is capable of running at a frequency above 100MHz. Even in the worst case scenario, where only one 128-bit block is encrypted, this implementation has a speedup of 43, compared to a pure software implementation running on a PowerPC at 300MHz.**

When encrypting a file of 16 kbits, the overhead of passing the data into the hardware (including the 1408 bits of the expanded key) is much less significant, thus a speedup of 569 times is obtained for the proposed design compared to the pure software implementation. This speedup corresponds to an increase on the encryption rate from 1.85 Mbits/s, for pure software, to 1057 Mbits/s in the polymorphic processor.

I. Introduction

In present days, almost every relevant communication system requires secure data transfer in order to maintain the privacy of the transmitted message; this message can be a simple email or a billion euro transaction between banks. In order to maintain the security of the communication channels, several encryption standards and algorithms exist, such as public key ciphers, symmetric ciphers and hash functions. Though public algorithms may offer an increased reliability, due mostly to the size of the keys, the computational requirements do not allow them to be efficiently used to encrypt the bulk of the data. To cipher large quantities of data, symmetrical ciphering algorithms are used. Because they are significant less demanding for an adequate security level. One of the emerging new NIST standards is the Advanced Encryption Standard (AES) [2].

Even though the symmetrical encryption algorithms are less computational demanding, they are still a critical part in purely software implemented applications. When implemented in hardware, the problem lays on the lack of adaptability to the constantly changing standards and the high cost for small scale productions. In order to use the best of both worlds, several reconfigurable proposals have been presented [3], based on FPGAs. However, most of these coprocessors are too specific and can not be easily adapted to other devices.

This paper proposes a AES encryption core that is used has a functional unit of a polymorphic processor, namely the MOLEN processor [1]. This approach allows the the AES core to be instantiated the same way has the equivalent software function, making it completely transparent to the software developer, that wishes to improve is system through the use of reconfigurable hardware.

The following section describes the structure of the AES encryption algorithm. The third section elaborates the proposed architecture for this algorithm to be used in the polymorphic processor, presenting a fine grain structure that mostly uses the FPGAs LUTs, and a memory based structure that takes advantage of the internal memory blocks (BRAMs) that exist in the newer FPGA devices. Section 5 presents experimental results for the developed core, on the Alpha Data: ADM-XPL development board with a Xilinx Virtex II Pro (xc2vp20) FPGA, using a 128-bit key. The last section finalizes this paper with some concluding remarks.

II. AES description

The AES is a standard of the NIST and uses the Rijndael encryption algorithm. It is becoming the replacing standard for the old, but still used, 3DES [4]. The AES algorithm is capable of using cryptography keys of 128, 192, 256 bits to encrypt and decrypt data blocks of 128 bits. Note that the Rijndael encryption algorithm also allows data blocks of 192 and 256, these however are not parte of the AES standard, and thus will not be mentioned on this paper.

As most of the symmetrical encryption algorithm, AES operations consists of byte substitution, bit permutation and the addition of the expanded Key, performed a predefined number of times, designated by rounds. These operations are performed through the usage of lookup tables to perform the byte substitution, column shifts and arithmetic operations in finite fields (addition and multiplications in $GF(2^8)$).

To better manipulate the 128 bits of the data block, they are represented by a matrix with 4 rows (r), each with N_b bytes, designated by the State array (S), were N_b is the dimension of the block in bytes divided by 4. Each column (c) has one byte of the input designated by $S_{r,c}$, with $0 \leq r < 4$ and $0 \leq c < N_b$. For this standard $N_b = 4$, i.e. $0 \leq c < 4$, resulting in a square matrix, as depicted in figure 1. The output is the State array after all rounds have been computed. For the AES algorithm, the length of the Cipher Key, is 128, 192, or 256 bits. The key length is represented by $Nk = 4$, 6, or 8, which reflects the number of 32-bit words in the Cipher Key. For the

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{0,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

Fig. 1: AES matrix organization.

AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr, where $N_r = 10$ when $N_k = 4$, $N_r = 12$ when $N_k = 6$, and $N_r = 14$ when $N_k = 8$. The number of rounds, Key length and Block Size in the AES standard id summarized in table I.

TABLE I: Key-Block-Round Combinations for AES.

	Key Length (Nk words)	Key Length (bits)	Number of Rounds (Nr)
AES-128	4	128	10
AES-192	6	192	12
AES-256	8	256	14

A. Encryption

As mentioned before the coding process consists on the manipulation of the 128-bit data block through a series of logical and arithmetic operations, repeated a fixed number of times. This number of rounds is directly dependent on the size of the cipher key. In the computation of both the encryption and decryption, a well defined order exists for the several operations that have to be performed over the data block. The encryption process is depicted in figure 2

State = in

AddRoundKey(State, key[0 *to* Nb−1])
for round= 1, round<Nr, round=round+1 **do**
  SubBytes(State)
  ShiftRows(State)
  MixColumns(State)
  AddRoundKey(State,key[round×Nb *to* (round+1)×Nb−1])
end for

SubBytes(State)
ShiftRows(State)
AddRoundKey(State,key[Nr×Nb *to* (Nr+1)×Nb−1])
out = State

Fig. 2: Pseudo Code for AES Encryption.

The next subsections describe in detail the operation performed by each of the functions used in figure 2, for the particular case of the encryption.

SubBytes()Transformation: The replacement of one set of bits by another is a non linear transformation, and is one of the most common operations in symmetrical encryption algorithms. In the Rijndael algorithm, this replacement is performed over a set of 8 bits (1 byte). Unlike the DES, this replacement can be described by a affine transformation (over $GF(2)$), as presented in equation 1.

$$b_i^{'} = b_i \oplus b_{(i+4)mod_8} \oplus b_{(i+5)mod_8} \oplus b_{(i+6)mod_8} \oplus b_{(i+7)mod_8} \oplus c_i \ ; \ 0 \leq i < 8 \quad (1)$$

where b_i in the n-th bit of the byte b(x) obtained from the State array. This bit permutation and transformation is performed over each byte individually. The c_i is the i-th of the value $\{01100011\}$.

ShiftRows(): The bytes in each row is shifted to the left by n bytes, with n being equal to the row number, for example $S_{1,0}S_{1,1}S_{1,2}S_{1,3}$ is transformed to $S_{1,1}S_{1,2}S_{1,3}S_{1,0}$. This operation is illustrated in figure 3.

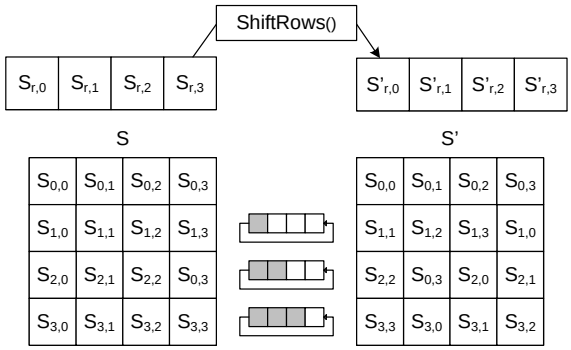

Fig. 3: AES ShiftRows.

MixColumns() Transformation: In this transformation each column is treated as a four-term polynomial over $GF(2^8)$ and multiplied modulo x^4+1 with a fixed polynomial $a(x)$, given by:

$$a(x) = 03x^3 + 01x^2 + 01x + 02 \quad (2)$$

resulting in the column given by:

$$\begin{aligned} S_{0,c}^{'} &= (02 \bullet S_{0,c}) \oplus (03 \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \\ S_{1,c}^{'} &= S_{0,c} \oplus (02 \bullet S_{1,c}) \oplus (03 \bullet S_{2,c}) \oplus S_{3,c} \\ S_{2,c}^{'} &= S_{0,c} \oplus S_{1,c} \oplus (02 \bullet S_{2,c}) \oplus (03 \bullet S_{3,c}) \\ S_{3,c}^{'} &= (03 \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (02 \bullet S_{3,c}) \end{aligned} \quad (3)$$

Figure 4 illustrates the MixColumns() Transformation.

AddRoundKey(): The final operation to be performed in each round is the addition (in $GF(2^8)$) of the respective Key with each column of the State matrix, simply performed by bitwise XOR. Each round Key consists of 4 32-bit words from the expanded Key (xK). The operation formalized in equation 4 is depicted in Figure 5, where $i = round \times N_b$.

$$[S_{0,c}^{'}, S_{1,c}^{'}, S_{2,c}^{'}, S_{3,c}^{'}] = [S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus [xK_{round \times N_b + c}] \ ; \ 0 \leq c < N_b \quad (4)$$

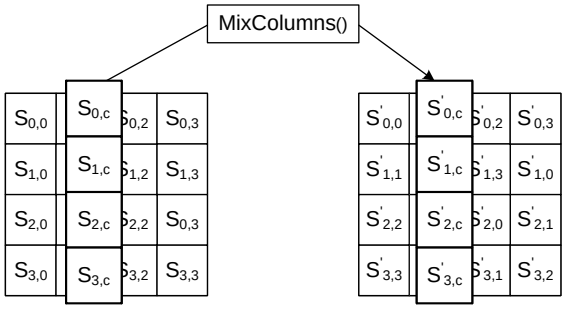

Fig. 4: AES MixColumns Transformation.

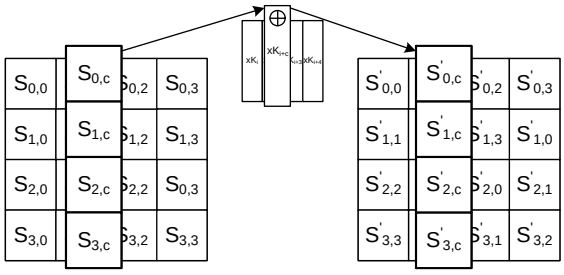

Fig. 5: AES key addition.

Key expansion: In this ciphering algorithm, the input Key has N_k 32-bit words, however $N_b(N_r + 1)$ 32-bit words are needed. Since $N_k < N_b(N_r + 1)$ the input Key has to be expanded. The resulting expanded key consists of a linear array of 4-byte words, denoted $[xK_i]$, with i in the range $0 < i < N_b(N_r + 1)$.

Each subkey is a 32-bit word, that in the case of the first N_k subkeys are directly obtained directly from the input key. The remaining subkey however, require some additional computation as described the pseudo code depicted in figure 6. This computation mostly consists in byte substitution as described for the SubBytes() transformation and a XOR operation with a constant $Rcon[i]$. Every N_k subkey calculation also requires an one byte left rotation.

for $i = 0$, $i < Nk$, $i = i + 1$ **do**
  sKey$[i]$ =word(Key$[4 \times i]$, Key$[4 \times i + 1]$, Key$[4 \times i + 2]$, Key$[4 \times i + 3]$)
end for

for $i = Nk$, $i < Nb \times (Nr + 1)$, $i = i + 1$ **do**
  if $i \mod$ Nk$= 0$ **then**
    temp=SubWord(RotWord(temp)) xor Rcon$[i/$Nk$]$
  else if (Nk>6) and $(i \mod$ Nk$) = 4$ **then**
    temp=SubWord(temp)
  end if
  sKey$[i]$ =sKey$[i-$Nk$]$ xor temp
end for

Fig. 6: Pseudo Code for AES Key expansion.

The SubWord is exactly the same as the SubBytes() transformation, in which every byte of the word is applied to the S-Box previously described. The function RotWord() takes a word [a0,a1,a2,a3] as input, performs an one byte left rotation, and returns the word [a1,a2,a3,a0]. The round constant word array, $Rcon[i]$, contains the values given by $[x^{i-1}, 00, 00, 00]$, with x^{i-1} being powers of x (x is denoted as 02) in the field $GF(2^8)$. The subkey $W[i]$ is calculated through the manipulation of the previous calculated subkey $W[i-1]$ using the previously described functions and finally with the bitwise XOR operation with the $W[i - N_k]$ subkey [5].

B. Decryption

In order to recover the data previously encrypted it is required to perform the inverse of the AES cipher, this process is identical the the encryption, however the operations performed in each subroutines different.

The structure of the inverse AES cipher is described in the pseudo code depicted in figure 7. Note that the expanded key for the decryption process is different than the one used in the encryption, although both are obtained (expanded) from the same original key [5].

State = in

AddRoundKey(State, key[Nr$\times$Nb to (Nr+1)$\times$Nb$-$1])
for round$= Nr - 1$, round≤ 1, round=round-1 **do**
  InvSubBytes(State)
  InvShiftRows(State)
  InvMixColumns(State)
  AddRoundKey(State, key[round$\times$Nb to (round+1)$\times$Nb$-$1])
end for

InvSubBytes(State)
InvShiftRows(State)
AddRoundKey(State, key[0 to Nb$-$1])
out = State

Fig. 7: Pseudo Code for AES inverse Cipher.

InvShiftRows: The InvShiftRows is the same as the ShiftRows with the difference that the rotation is performed to the right instead of the left.

InvSubBytes: The InvSubBytes is the inverse of the byte substitution performed on the encryption (SubBytes). This is obtained by applying the inverse of the affine transformation described in equation 1, followed by taking the multiplicative inverse in $GF(2^8)$. The lookup table values required to perform this transformation (substitution) are presented in [5]. This table consists on a 8-bit input to an 8-bit output transformation.

InvMixColumns: The InvMixColumns is the inverse of the MixColumns transformation previously presented for the encryption.

Once again the columns of the S state are considered as polynomials over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, which is the inverse of $a(x)$, given by:

$$a(x) = 0bx^3 + 0dx^2 + 09x + 0e \quad (5)$$

resulting in the column given by:

$$
\begin{aligned}
S'_{0,c} &= (0e \bullet S_{0,c}) \oplus (0b \bullet S_{1,c}) \oplus (0d \bullet S_{2,c}) \oplus (09 \bullet S_{3,c}) \\
S'_{1,c} &= (09 \bullet S_{0,c}) \oplus (0e \bullet S_{1,c}) \oplus (0b \bullet S_{2,c}) \oplus (0d \bullet S_{3,c}) \\
S'_{2,c} &= (0d \bullet S_{0,c}) \oplus (09 \bullet S_{1,c}) \oplus (0e \bullet S_{2,c}) \oplus (0b \bullet S_{3,c}) \\
S'_{3,c} &= (0b \bullet S_{0,c}) \oplus (0d \bullet S_{1,c}) \oplus (09 \bullet S_{2,c}) \oplus (0e \bullet S_{3,c})
\end{aligned} \quad (6)
$$

Note that this polynomial multiplication is more computational demanding than the one required by the encryption, having more multiplication and the constants to be multiplied have more non zero bits.

Key expansion: The key expansion algorithm for the decryption is identical to the key expansion algorithm for the encryption, also resulting in a expanded key with $N_b(N_r+1)$ 32-bit words [2]

III. Implementation

This section presents the implementation details of the several components of the AES core as well as the connection to the PowerPC (used in this polymorphic processor). This implementation has in mind the Xilinx virtex II pro FPGAs, however its VHDL description can be used to implement this core in any reconfigurable device with an identical structure. The coarse grain architecture though, requires the existence of ROMs with a 8 bit input and a 32 bits output (with 1024 bytes of addressable memory).

A. AES core

The most linear way to implement in hardware the AES core, consists on designing each of the subroutines presented in figure-2 in hardware and connect them sequentially. Each loop of the for cycle is designated by round, the amount of rounds required depends on the size of the key and can be 10, 12 or 14 for 128-bit, 192-bit or 256-bit key respectively. It can be observed in the pseudo-code depicted in figure 2 that one last round is necessary. However this last round is simpler than the above, since it does not required the mixcolumn operation.

The first operation to be performed in the AES encryption is the addition of the corresponding expanded keys to the initial data block matrix. After this initial addition, the rounds have to computed (sequentially). Each round consists of a byte substitutions followed by row shifts and column mixing and finally the resulting intermediate data matrix is added to the corresponding subset of the expanded key. Finally to obtain the encrypted data the last round has to be computed, by performing byte substitutions, row shifts and the key addition.

Byte substitution: As described in section II, the byte substitution performed in the Rijndael encryption algorithm is given by a bit wise arithmetic function. The result of each bit depends on the value of the all byte.

This calculation can be performed either by a lookup table with an 8 input (an 256 memory block) or in hardware logic.

Even though a simple logical expression exists, the usage of look up tables may be more advantage when implementing on a FPGA, due to the existence of LUTs in this type of device.

Row shift: The shifting of rows of the state matrix is performed by a fixed byte reordering, which has no direct hardware requirements.

Column mixing: The mix column consists on calculating the new value for a given byte, based on the value of all the bytes on that column. Meaning that to calculate the byte in the matrix position $(1,2)(S_{1,2})$ all the values of column 2 have to be used ($S_{0,2}$,$S_{1,2}$,$S_{2,2}$,$S_{3,2}$).

This column is considered with a polynomial representation over $GF(2^8)$ and is multiplied modulo x_4+1 with the fixed polynomial $a(x)$ ($a(x) = 03x^3 + 01x^2 + 01x + 02$). The partial multiplications over the Galois field 2^8 is performed by a simple bitwise logical AND operation. The additions is also performed over the 2^8 Galois Filed ($GF(2^8)$), which means that the addition can be performed by a bitwise XOR logical operation, without the need for carry propagation. However, the multiplication of two bytes results in one number with more than 8 bits. In order to obtain a values with 8 bits, the result is replaced by the remainder polynomial, that in the case of Rijdael is given by the irreducible polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1. \quad (7)$$

Thus when ever the result of the multiplication has more than 8 bits the hexadecimal value $11B$ has to be subtracted, which in $GF(2^8)$ is the same as doing an XOR between the less significant byte of the result and the hexadecimal value $1B$. To better illustrate this multiplication the following example depicts the multiplication of the byte $D7$ by 3 (values in hexadecimal representation).

$$1 \times D7 = D7 \quad (8)$$

$$2 \times D7 = 1AE \Rightarrow 1AE - 1B = B5 \quad (9)$$

$$
\begin{aligned}
3 \times D7 &= (2+1) \times D7 = 2 \times D7 + 1 \times D7 \\
&= B5 + D7 = 62
\end{aligned} \quad (10)
$$

The conditional addition of the value $1B$ can be implemented by 4 XOR gates (which is the number of bits with the value 1 in $1B$). Figure 8 depicts the architecture used to compute the multiplication. Note that the multiplier used to compute the multiplication by 3 also produces the value of the multiplication by 2.

Observing equation 3 it can easily be recognized the locality of the data. In other words the values of column c depend only on the values of column c, these locality can significantly improve the performance of the implemented hardware, especially on reconfigurable devices such as FPGAs were the routing hardware is limited. Moreover, note that the value on each column of equation 3 is dependent on only one of the state ($S_{i,c}$) values, thus only one multiplier has to be used per each data byte, since one multiplier automatically calculates the multiplication by 2 and 3.

The final polynomial addition is performed by a a tree of XOR gates, since in $GF(2^8)$ no carry propagation is required. This addition structure is depicted in figure 9.

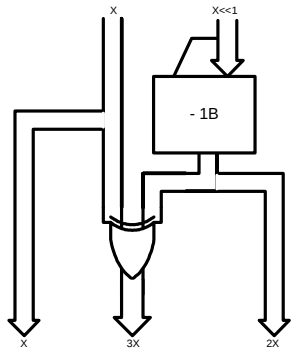

Fig. 8: $GF(2^8)$ multiplication architecture for encryption

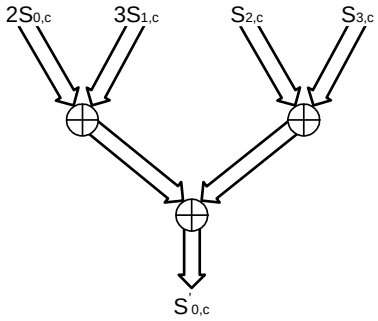

Fig. 9: Polynomial addition in $GF(2^8)$

Key addition: The key addition is also performed over the $GF(2^8)$, as such it is easily implemented by one XOR gate per each par of bits to be added (one from the data matrix and the other from the expanded key).

AES decryption core: In order to retrieve the original data previously encrypted, the data undergoes a process identical to that of the encryption (see pseudo code of figure 2). The main differences in the decryption computation lays on the byte substitution and on the polynomial equation used in the column mix.

Unlike the encryption, the byte substitution transformation has no apparent boolean expression, thus it has to be implemented by a look up table [2].

In the inverse column mix, the process is exactly the same as in the encryption, the only difference lays in the coefficients values. While in the encryption these coefficients result in a small hardware structure, since at most, only 2 bits are equal to 1 (the multiplications constants are $1, 2$ and 3), in the inverse column mix the coefficients $(9, b, d, e)$ usually have 3 bits at 1. The method used to compute the multiplications values in the $GF(2^8)$ is the same, for example to calculate the multiplication by the constant b, the multiplication by $8, 2, 1$ is first computed and only then added ($b = 8+2+1$) to obtain the value b. The multiplication architecture for the inverse column mix multiplier is presented in figure 10.

Implementing the AES cipher in memory banks: The hardware implementation previously presented describes a fine grain implementation of the AES cipher. However the recent

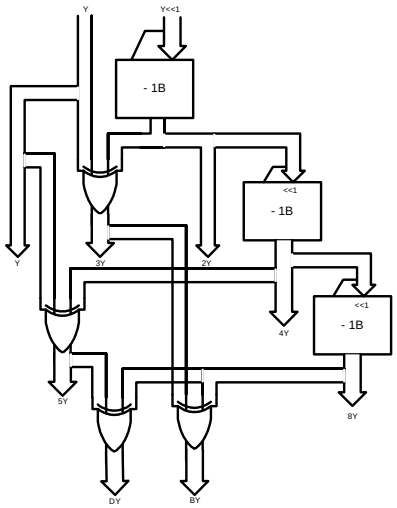

Fig. 10: $GF(2^8)$ multiplication architecture for decryption

programable devices, such as the Xilinx Virtex II Pro FPGAs, possesses versatile and fast memory banks in sufficient number that allow for more coarse grain architectures.

While in a fine gain architecture the computation for the byte substitution and column mix multiplication have to be implemented in different and specific hardware structures, as depicted in figure 11, in a memory bank based coarse grain architecture these two units, along with an hardwired row shift, can be merged together and easily mapped into, an 8 bit input and 32 bit out output, memory bank. The

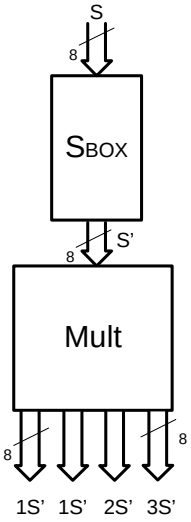

Fig. 11: Fine grain column computation

memory bank receives one byte from the state matrix and outputs that value multiplied by the 4 coefficients after the byte transformation, $\{1, 1, 2, 3\}$ for the encryption and $\{9, c, d, e\}$ for the decryption. Since, in order to calculate each column four of these outputs are required (see equations 3 and 6) four memory banks would be needed. However, the memory

banks available in these FPGAs have a dual input and dual output ports, and since all the memory banks perform the same computation one FPGA memory bank can be used to implement 2 AES coarse grain memory banks, as depicted in figure 12 for the decryption case.

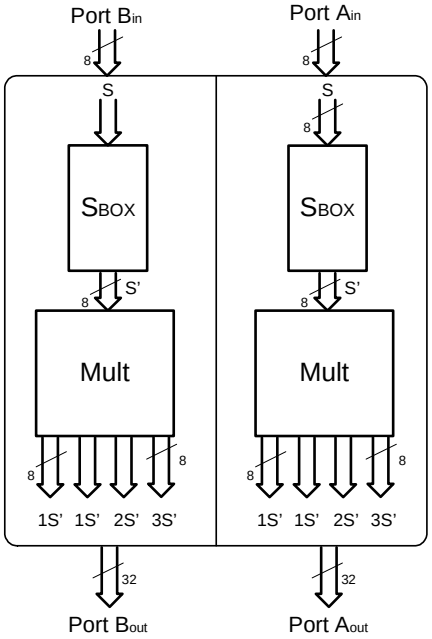

Fig. 12: Coarse grain column computation

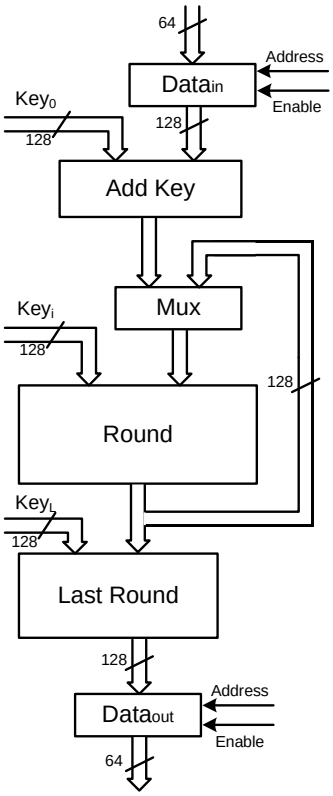

Fig. 13: AES Core

This coarse grain implementation has the potential to increase the architecture performance as well as to reduce the device occupation, since it uses more of the available hardware.

The full core processor: The architecture described so far has been for only one round, however in the full AES cipher the data has to be processed several times, i.e. several computation rounds. This can be done either by having each round implemented in its own hardware structure (fully unrolled), which implies a significant amount of hardware, or the hardware used to implement one round can be reused to compute the rest of the rounds (folded) as depicted in figure 13. Partially rolled implementations can also be used.

The computational flow of the full AES cipher, depicted in figures 2 and 7 (described in the previous section), also include an initial key addition and a simplified final round that does not perform the column mix (no multiplication). Both the prologue and the epilogue have their own hardware structures, resulting in the architecture depicted in figure 14. The expanded keys used in the main rounds (sets of 128 bits) are given according to the round being executed at that given cycle.

B. Data transfer

While in a fully software processor the data is passed through the stack when a software function is called, in the Molen processor when a hardware function is called, the data is passed through a special register designated by exchange register (XReg). The input data for the AES cipher processing core is not only the location of the data to encrypted, other values also have to be uploaded. The processing core needs to know how many rounds have to be computed (depending if the key has 128, 192 or 256 bits), it also has to know the end address for the data to be ciphered, since more than one data block may need to processed. Finally the begin and end address of the expanded key also has to be passed. With the number of rounds the end address of the key could be computed, however, when the same key is being used for more than one time it does not have to be uploaded again to the AES core. In these situations the end address can be the same as the begin address, and thus no new key will be loaded. Note that the internal registers of the AES core work as static variables, meaning that there previous state is maintained unless a new value is loaded.

As in a software function the, bulk of the data is obtained directly form the data memory region. The AES core starts by reading the key into the key register, this process requires some cycles, since the memory is accessed in blocks of 64 bits and the expanded key has at least 1408 (up to 1920 bits for a 256-bit key).

Although much smaller than the expanded key, the data block (128 bits) is still bigger than the length of the main data bus (64 bits), thus the data block as to be read and written in multiple cycles. This implies that additional hardware must exist to store and reassemble the each of the 64 bits of the data block. This is done by a small bank register with a single

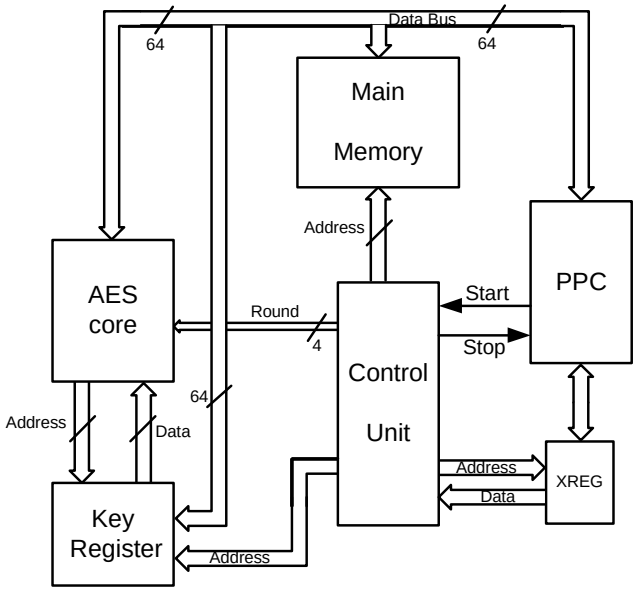

Fig. 14: AES cipher architecture

addressable 64-bit input and one 128-bit output, for the data block reception, and identically for the processed data block, as depicted in figure 14.

After this initial setup stage the AES core enters the burst mode, were it reads one data block (128 bits) after the other, in between every data block reading a processed data block is written back to the memory. Since the memory only has one data port, a data block reading can occur at the same time as a data block write.

C. Key register

The key associated to the cipher can not be used directly, this (128, 192 or 256-bit) key as to be expanded. For efficiency reasons, and taking into account that usually this expansion is only performed sporadically, it is computed in a fully software function and stored in the data memory. When the data blocks are being ciphered this expanded key has to be available, and due to its dimension (10 times bigger than the data block), it has to be stored locally in the AES core, otherwise a crippling memory access overhead would occur.

On a fully unfolded version of this AES core, the all expanded key has to be available at all times, this can be implemented by a register bank with an input length equal to the size of the main memory data bus and a output port with the size of the expanded key (that in the case of a 256-bit key corresponds to 1920 bits).

On the fully rolled AES core only 128 bits of the expanded key have to be available, thus a register bank can be implemented with an input port the size of the main memory data bus and a output port with 128 bits. In this case, the register has to be indexed according to the round being executed at the time. This type of addressable register bank can be directly mapped to a small memory bank. In the case of the used FPGA, 4 internal memory banks have to be used, not due to the lack of space in an single memory bank, but due to the fact that these internal memory banks have output port with at most 32 bits.

Even in the fully rolled AES core, depicted in figure 13, the first and last 128 bits of the expanded key are outside the main loop, and in some cycles are accessed simultaneously with the ones in the main loop. Due to this simultaneous access, these values have to be stored in separate registers. The key storing unit is depicted in figure 15

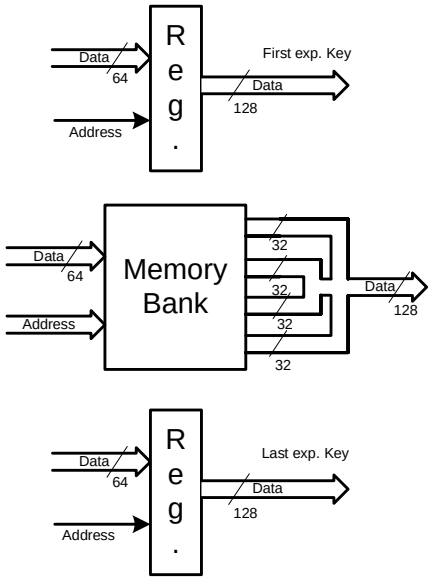

Fig. 15: Expanded key register

D. Control unit

The architecture here described requires a significant amount of control signals, which are generated in the control unit. This control unit is composed not by one, but by two state machines. A main state machine and a secondary one designated by write state machine responsible for the writing of the processed data block and for signalling the PowerPC of the conclusion of the process (function).

After filling the XREG with the input parameters for this hardware function (the equivalent of filling the stack memory in a software routine) the PowerPC signals the AES core (namely the control unit) to start via the $start$ signal.

Main state machine: This signal is directed to the main state machine, that starts by reading the begin and end addresses were the expanded key is located, from the predefined addresses in the XREG. These values are stored in dedicated registers. The begin address is sequentially incremented and its value used to address the main memory. The output of the memory (the expanded key value) is stored in the expanded key register. The state machine also has a dedicated counter used to address the expanded key register. The increments of the memory depend on the length of the memory data bus, in this case the address is incremented by 8 (64 bits$= 8 \times 8$ bits). When the incremented address becomes equal to the given end address the state machine switches to the next state.

In the following 2 states the begin and end data addresses are copied from the XREG to the two previously mentioned registers, as well as to two additional address registers that will be used by the write state machine to address the main memory to write the processed data blocks.

At this moment the main state machine synchronizes it self with the write state machine, by sending a signal for this state machine to start (through a single bit value).

From this moment on, the main state machine enters a loop in which it reads one data block from the memory and waits until a new data block can be read. During this wait state, addresses are generated for the expanded key register, depending on the current round.

When the incremented address becomes equal to the data end address it exits the loop and returns to the initial state were it waits for a $start$ signal from the PowerPC. The $stop$ signal that indicates the conclusion of the encryption process is generated by the write state machine.

Write state machine: The write state machine is responsible for controlling the writing of the processed data block back to the main memory, in order for the writing to be synchronized with the data block read from the main memory, two wait states have to be implemented.

When the synchronization signal is received from the main state machine, a non repeatable waiting loop is performed. This waiting period depends on the time required for the pipeline to be filled, which varies according to the depth of the pipeline. Only when the pipeline is full can the main writing loop begin.

After the initial synchronization, this state machine enters the main loop. On it, a inner waiting loop is executed in order for the data to be processed or for a new data block to be read by the main state machine. When ready it jumps to the writing state, were the processed data is written to the main data memory and the writing address incremented.

The main loop is stopped when the the incremented address equal the end address. When that occurs the $stop$ signal is sent to the PowerPC signaling the end of the (hardware) function and the state machine returns to the initial state, waiting for the synchronization signal of the main state machine.

Control unit characteristics: The implementation of the control units as a dual state machine allows a better synchronization between the writing and reading of data blocks. It also facilitates its modification for different pipeline depths, different unrolling architectures, different lengths of the main memory as well as different memory architectures (i.e. if it has a common read/write address port a extra cycle has to exist between each read/write sequence).

Since the memory write state machines is synchronized with the main state machine through a single bit line, it allows the two state machines to be in different locations, and thus closer to the hardware they have to control, allowing for an optimization on the routing, which can be critical in a FPGA hardware implementation.

This control unit has been designed in a parameterizable fashion, as such it is possible to alter its main parameters by changing the constant values defined in a VHDL file. This way, major changes can be made by different designers, with a complete abstraction of the control machine architecture.

IV. Experimental results

In order to test the resulting architecture, the AES core prototype (for a 128-bit key) has been implemented in a Xilinx Virtex II Pro (xc2vp20) on an Alpha Data: ADM-XPL development board using the ISE (6.3) and SDK (6.3) tools from Xilinx . This core has been developed to be integrated in the MOLEN polymorphic processor [1], which uses the PowerPC embedded in the FPGA and capable of running at maximum a frequency of 250 MHz.

Table II shows the synthesis results of the AES core for the fine grain (fg) and for the memory based (mb) architecture, for both encryption and decryption. These results have been obtain for the fully rolled version of the AES cipher core.

TABLE II: AES core implementation results

architecture	Slices	BRAMs	Time(ns)
fg - encryption	3224 (35%)	4 (4%)	8.45
fg - decryption	3498 (38%)	4 (4%)	8.86
mb - encryption	1879 (20%)	12 (13%)	7.51
mb - decryption	1933 (20%)	12 (13%)	7.51

These results show that a compact architecture can be derived for this new cipher algorithm and at the some time with a working frequency that would allow a throughput of approximately 1.7 Gbits/s at a operating frequency of 133 MHz, however the system as a bottleneck in the main memory operating frequency that can only run at 80 MHz. At this frequency and taking into account that each block is computed in the main round for 10 cycles, the system has an estimated throughput of 1.02 Gbits/s ($= 80MHz/10 \times 128$ bits), with both the architecture types (fine grain and memory based). These values disregard the setup time required for the AES core initialization.

From table II it is clear that the memory based implementation results in a better usage of the FPGA resources, since the number of used BRAMs (13%) and slices (20%) is more homogeneous than the fine grain implementation, where only 4% of the BRAMs are used, requiring more that one third of the available slices.

In order to increase the throughput, the main loop of AES core can be unrolled, which obviously represents an increase on the required hardware. Table III presents the additional FPGA resources necessary to include one additional round.

TABLE III: AES core additional round requirements

architecture	Slices	BRAMs
fg - encryption	1337 (14%)	4 (4%)
fg - decryption	1642 (17%)	4 (4%)
mb - encryption	221 (2%)	12 (13%)
mb - decryption	194 (2%)	12 (13%)

From values in table III it can be concluded that some loop unrolling can be performed with small additional increase in

the cost. However since the memory throughput is at most 6.4 Ghz ($100MHz \times 64bit$) and due to the fact that every read block has to be written back, the system maximum ciphering rate is of 3.2 GHz. Thus, by unfolding the AES core main loop once the maximum processing rate is obtained. With the main loop unfolded once the AES core processing rate is doubled, having a expected ciphering rate of about 2.5 Ghz.

In order to compare the speedup archived by this core, the algorithm has been executed in a fully software implementation on the PowerPC at his maximum frequency of 300 Mhz and in the hybrid implementation using the MOLEN processor also using the PowerPC. In order to accurately measure the performance, the throughput was obtained by counting the PowerPC clock cycles (at 300MHz), that is one third of the FPGA clock cycle (at 100MHz). Experimental results depicted in table IV show a clear improvement when the AES core is used.

TABLE IV: AES performances

	Hardware		Software	
Bytes	Cycles	Throughput	Cycles	Throughput
16	560	76 MHz	24216	1.76 MHz
512	1800	759 MHz	738952	1.85 MHz
16k	41480	1057 MHz	23610504	1.85 MHz

While the obtained speedup is just of 43 times (table V) when only one 128-bit data block is encrypted, due to the overhead to transfer the expanded key (1408 bits), a speedup of 569 is accomplished for a file with 16kBytes. Note that the overhead of the expanded key transference is already no very significant, specially considering that most private key encryption applications usually have to encrypt files with significantly large dimensions. For specific applications that have real time demands and require to send small amounts of data as soon as they are available (for example a text console), an higher throughput can be reach, since the expanded key only has to be uploaded to the AES core once. In this case a throughput of 98 Mbits/s can be archived, instead of 76 Mbit/s for a single 128-bit data block.

TABLE V: AES core Speedup

Bytes	Speedup
16	43
512	410
16k	569

All these values were obtained for a FPGA with clock frequency of 100 MHz, due to the memory bottleneck. With a faster main memory, even higher speedups can be reached since the AES core is capable of a 133 MHz working frequency.

V. Conclusion

The hybrid hardware-software implementation of the AES encryption algorithm on the MOLEN polymorphic processor presented in this communication shows to be clearly advantageous. In practical application, where large blocks have to be encrypted and decrypted, this implementation is capable of speedups of 450 times with a significant small amount of reconfigurable hardware (less than 2000 slices), 20% on a xc2vp20 Virtex II pro FPGA. The type of integration of this AES core on the MOLEN processor allows for a expedite integration in any type of software application making it possible to be used on the new 1 Gbit ethernet applications, with the 1 Gbits/s processing rate obtained from a speedup of 569. The control unit structure and design allow for an easy adaptation of the AES core to different specifications, such as different memory architectures or even when loop unrolling is applied to the hardware structure to achieve a higher throughput.

Future work in this core will include merging the encryption and decryption core in one single unit, that in the coarse grain architecture may result in practically null area increase, since only half of the addressable memory in the BRAMs is being used. Another optimization may lay on the calculation of the last round with the same hardware has the main rounds, which in the fully rolled version will result in a significant hardware reduction.

References

[1] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363– 1375, November 2004.

[2] J. DAEMEN and RIJMEN, "The design of rijndael. aes-the advanced encryption standard," *Springer-Verlag*, 2002.

[3] F. Rodriguez-Henriquez, N. Saqib, and A. Diaz-Perez, "4.2 Gbit/s single-chip FPGA implementation of AES algorithm," *Electronics Letters*, vol. 39, pp. 1115–1116, July 2003.

[4] NIST, "Data encryption standard (DES), FIPS 46-2 ed," tech. rep., National Institute of Standards and Technology, December 1993.

[5] J. Daemen and V. Rijmen, "Advanced encryption standard (AES) (FIPS 197)," technical report, Katholijke Universiteit Leuven/ESAT, November 2001.