# An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm

G.P. Saggese[1], A. Mazzeo[1], N. Mazzocca[2], and A.G.M. Strollo[1]

[1] University of Naples Federico II, Via Claudio 21, 80125 Napoli, Italy
[2] Second University of Naples, Via Roma 29, 81031 Aversa, Italy
{saggese,mazzeo,n.mazzocca,astrollo}@unina.it

**Abstract.** In October 2000 the National Institute of Standards and Technology chose Rijndael algorithm as the new Advanced Encryption Standard (AES). AES finds wide deployment in a huge variety of products making efficient implementations a significant priority. In this paper we address the design and the FPGA implementation of a fully key agile AES encryption core with 128-bit keys. We discuss the effectiveness of several design techniques, such as accurate floorplanning, the unrolling, tiling and pipelining transformations (also in the case of feedback modes of operation) to explore the design space. Using these techniques, four architectures with different level of parallelism, trading off area for performance, are described and their implementations on a Virtex-E FPGA part are presented. The proposed implementations of AES achieve better performance as compared to other blocks in the literature and commercial IP core on the same device.

## 1 Introduction

Symmetric-key block ciphers are important in many cryptographic systems. Individually they provide confidentiality which means keeping information secret from all but those who are authorized to see it. As a basic building block, they allow construction of pseudo-random number generators, stream ciphers, message authentication code, and hash functions. These primitives have a crucial role in many real-world applications. In October 2000 the National Institute of Standards and Technology (NIST) chose Rijndael algorithm [1] as the new Advanced Encryption Standard (AES) [2]. This standard is become effective on May 2002. The use of AES and the AES replacement of DES and triple DES, is encouraged to provide the desired security of electronic data in commercial and private organizations.

Even if most new algorithm designs cite efficiency in software as a design objective, the software implementations of symmetric algorithms are often computationally expensive. Therefore in many applications the use of hardware-based solutions has become unavoidable in order to meet high performance requirements. Reconfigurable devices, like Field-Programmable Gate Arrays (FPGA),

P.Y.K. Cheung et al. (Eds.): FPL 2003, LNCS 2778, pp. 292–302, 2003.

are a promising alternative for the implementation of block ciphers, since they include many advantages of software implementations (like algorithm agility, algorithm modification capability [9], and cost efficiency) with physically security and high throughput of an Application-Specific Integrated Circuits (ASIC). In order to achieve maximum performance on FPGA, designs making use of architecture-specific features are required, which means that circuit implementations are not portable between different FPGA technologies.

In this paper we address the design and the FPGA implementation of a fully key agile AES encryption core with 128-bit (128$b$) keys. We discuss the effectiveness of several techniques, such as accurate floorplanning, the unrolling, tiling and pipelining transformations (also in the case of feedback modes of operation) to explore the design space, allowing to tradeoff area for performance. The proposed AES blocks achieve better performance (with respects to area and throughput metrics) than other implementations on the same Xilinx device, available in the technical literature and as commercial IP cores. The application of the above-mentioned techniques allow an effective design space exploration, since our least area (iterative) design requires 446 Virtex-E slices and 10 Select BlockRAM delivering 1 gigabit per second (Gbps) encryption rate, and our fastest AES block (fully unrolled and deeply pipelined) reaches a throughput of 20,3 Gbps.

The rest of the paper is organized as follows. In Section 2 we describe the AES algorithm and the block cipher modes of operation. Section 3 addresses main issues related to the design and the implementation of AES and discusses our solutions. Section 4 describes four architectures differing in unrolling and pipelining level and presents their floorplan aware implementations. Section 5 reports performance results and analyzes area vs throughput trade-offs, with respect to other academical and commercial implementations. Finally, Section 6 concludes the paper with some final remarks.

## 2 AES Encryption Algorithm and Modes of Operating

The AES algorithm selected by NIST as the successor of the DES encryption algorithm is Rijndael [1]. In AES standard [2] the length of the data block is always equal to 128$b$ and so the parameter $Nb$, which represents the number of 32$b$ words composing a data block, is 4. As far as different security requirements are concerned, the key length for AES can be chosen as either 128$b$, 192$b$, or 256$b$ long, and the corresponding parameter $Nk$ is 4, 6, or 8 respectively. The pseudo-algorithm for the AES encipherment and for the `KeyExpansion` procedure with a 128$b$ or 192$b$ key are reported in the left and right boxes of Fig. 1, respectively. The main computation of AES encryption is a loop repeated $Nr$ times, where $Nr$ is equal to either 10, 12, or 14, based on the key size. The cipher algorithm has the goal of obscuring (in a reversible manner) the plain text with a set of $Nb(Nr + 1)$ 32$b$ words `w` (named *sub-keys*) derived by the expansion of the `key`. AES interprets the incoming data and the intermediate result (`state`) as a 4×4 array of 8$b$ words (`byte`). The processing of a 128$b$ plain text starts

```
Cipher (byte in[4*Nb],
  word w[Nb*(Nr+1)], byte out[4*Nb])
begin
  byte state[4,Nb] = in
  AddRoundKey(state, w[0, Nb-1])          } Initial Round
  for round = 1 step 1 to Nr-1
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)                     } Round
    AddRoundKey(state,
    w[round*Nb, round*Nb+Nb-1])
  end for
  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state,                      } Final Round
    w[Nr*Nb,Nr*(Nb+1)-1])
  out = state
end
```

```
KeyExpansion (byte key[4*Nb],
  word w[Nb*(Nr+1)])
begin
  word temp, i = 0
  while (i < Nk)
    w[i] = (key[4*i], key[4*i+1],
      key[4*i+2],key[4*i+3])
    i = i+1
  end while
  while (i < Nb*(Nr+1))
    temp = w[i-1]
    if (i mod Nk = 0)
      temp = SubWord(RotWord(temp))
        xor Rcon[i/Nk]
    end if
    w[i] = w[i-Nk] xor temp
    i = i+1
  end while
end
```

**Fig. 1.** The AES `Cipher` algorithm in the case $Nb = 4$ and the `KeyExpansion` procedure particularized for $Nk = 4$ and 6, in the left and right box, respectively.

copying it into the `state`. The `state` is mixed with the first part of the sub-keys by means of an `AddRoundKey` step (*InitialRound* in Fig. 1). Then the `Round` transformation, involving the four basic steps of Fig. 1, is applied $Nr - 1$ times. At last the *FinalRound* block, which differs slightly from the *Round* since it does not include `MixColumns`, is executed. The basic operations which compose AES are: 1) `SubBytes` – each byte of the `state` is transformed using an s-box function which is composed of two transformations over GF($2^8$), namely an inversion and an affine transformation (i.e. a matrix per vector multiplication); 2) `ShiftRows` – each row of the matrix `state` is circularly shifted with a row-dependent amount of positions; 3) `MixColumns` – is a multiplication over GF($2^8$) of each column for a constant, giving the corresponding column of the updated `state`; 4) `AddRoundKey` – is a bitwise modulo-2 addition, i.e. an addition over GF($2^8$), of a sub-key to the state. The AES algorithm takes the cipher key and performs a `KeyExpansion` procedure to generate the linear array of $Nb(Nr + 1)$ sub-keys, which are used in the $Nr + 1$ steps of the cipher algorithm. In the `KeyExpansion` procedure of Fig. 1 `RotWord` is a circular rotation of the bytes in a word and `SubWord` applies an s-box transformation to each byte of a $32b$ word. The constant `Rcon[h]` is $[\{02\}^{h-1}, 00, 00, 00]$ where the power is considered over GF($2^8$). Several modes of operation for use with an underlying symmetric key block cipher algorithm are recommended by NIST [4] and are widely employed. The main goal of these modes of operation is twofold [3]: 1) improving the security of block ciphers avoiding repeated inputs from being encrypted to the same value. To achieve this goal, the output of the block cipher is fed back, introducing a dependence of current output upon previous inputs. 2) Providing different security features (such as hashing, integrity check, etc.) using the block cipher as a building block. The main modes of operation are: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). Specific information about the security

| | ECB | CBC | CBC-MAC | CFB | OFB | OCB | CTR |
|---|---|---|---|---|---|---|---|
| **Sender** | Enc | Enc | Enc | Enc | Enc | Enc | Enc |
| **Receiver** | Dec | Dec | Enc | Enc | Enc | Dec | Enc |
| **Feedback** | No | Yes | Yes | Yes | Yes | No | No |

**Fig. 2.** Block cipher modes of operation.

properties of these modes of operation, with respect to errors, manipulations, and substitutions, and the associated schematics, can be found in [3] [4]. The Fig. 2 reports, for each mode of operation, whether encrypter (Enc) or decrypter (Dec) is needed in transmitting and receiving phase, and the feedback loop in the processing flow of a data stream is present.

## 3 Main AES Design Choices

**Unrolling, tiling and pipelining AES.** – Many cipher algorithms are based on a basic looping structure (Feistel or other substitution-permutation network) whereby data are iteratively passed through a transformation, namely the "round" function. Three techniques are known in the technical literature to exploit the intrinsic parallelism of this kind of algorithm at different levels providing architecture with different performance: the unrolling, the tiling and the pipelining transformation. The *unrolling* [9] consists in allocating the body of different instances of a loop some number of times (called the unrolling factor $U$) and iterates the loop by step $U$. Referring to $U$, the number of instanced loop bodies, it is possible to distinguish an iterative (serial) architecture, a partially unrolled or a completely unrolled one. The *iterative* architecture consists in the instantiation of a single body of the loop. This block is fed back through a mux, in order to realize successive iterations of the loop. This approach can minimize the hardware requirement giving low throughput. In an *unrolled* architecture the algorithm is implemented by allocating $U$ rounds as a single combinatorial logic block. This approach requires more area and does not necessarily increase the throughput but enables pipelining, as it is shown in the following of the paper. The *tiling* [5] transformation replicates the instances of a block in order to increase the functional parallelism. Finally, the *pipelining* increases the number of blocks of data that are being simultaneously operated upon, inserting registers to store the partial results.

In the recent past, some authors [5] asserted that, in order to increase the throughput, it is better tiling a serial AES block than unrolling it. They justified this remark noting that: 1) an unrolled architecture is faster than the iterative one in the encryption time of a block, but this difference is often negligible; 2) the time saving comes with a greatly increase of the area. We agree with this analysis (that can also be supported by the quantitative measurements presented in [9]), but we would emphasize that tiling has different flaws with respect to unrolling as a throughput increasing technique. First of all, the unrolling enables further increase of pipelining levels improving the throughput, and this can justify the

extra area. Second, when the design is tiled, additional logic (with corresponding area and time penalty) is usually involved: demultiplexing logic to feed each one of the blocks, and a block collecting the results. An unrolled and pipelined architecture does not require any other additional logic than pipeline registers. In this paper we propose (see Section 4) several AES implementations, differing in the number of rounds actually instanced and in the level of pipelining. These demonstrate that the throughput per area of an iterative AES block is lower than the one of an unrolled and pipelined version of AES, even without considering the extra logic to route the data in a tiled architecture.

**Pipelining and Feedback Loops** – As we already mentioned, in many real-world applications, cipher modes of operation are employed. The pipelining can be effectively used for applications requiring modes of operation not relying on feedback (such as ECB, OCB, and CTR) to enhance the performance. Some of these modes imply feedback (see last row of Fig. 2). The feedback introduces a loop carried dependence, i.e. the processing of a block can start only if the previous block has been processed, and this prohibits a pipelined implementation. For this reason, some cipher modes, such as like interleaved CFB or CFB-k [4], [3], have been proposed to allow security and high encryption rate when implemented in hardware. In the applications requiring feedback modes (such as CBC, CFB, and OFB), a method of increasing the throughput is to pipeline the cipher even in presence of feedback, and to interleave multiple data streams keeping full the pipeline This technique [6] is called *innerroundpipelining* (or c-slow technique). Please note that it is possible to resort to the same technique also in an iterative architecture, with or without feedback cipher mode. As general rule in both cases, if the pipeline is composed by $N$ stages, $N$ different data flows should be available. The availability of many independent data flows is a common scenario in encrypted Internet router whereas multiple routes are active and have to be encrypted or authenticated. Since an (iterative or feedback) AES block can be pipelined, provided that a sufficient number of data flows is available, in the following of this paper we focus on the application of pipelining on the AES algorithm.

**Decryption and Key-Length** – In this paper we focus on the design and the implementation of an AES encrypter for two main reasons: 1) in many communication systems only the encipherment can suffice (see first and second row of Fig. 2); 2) the AES decryption process is roughly the same as the encryption one. In fact, the same loop structure of the encryption and similar steps (`InvByteSubs`, `InvShiftRows`, etc.) are used in the decryption and thus the same solutions to architectural and implementation issues we propose can be extended to an AES decryption core.

As far as the key-length is concerned, we consider the design of an AES encryption block for 128$b$ keys. However in the same way, it is possible to design AES encryption cores with 192$b$ and 256$b$ keys, since they differ in the number of rounds and in a slightly different `KeyExpansion` algorithm.

**On/Off-Line Subkey Generation** – The sub-keys for encipherment/decipherment can be either stored in a local memory (*stored-key* approach) or

generated concurrently with the encryption/decryption process (*key-agile* approach). The stored-key approach requires a preprocessing phase every time the key is changed, and it needs a (quite large) memory block for the sub-keys. The key-agile approach allows the block cipher to work at full speed, even if the key is changed, and saves the extra memory for the sub-keys. It is worth noting that in an unrolled and pipelined version of a key-agile cipher, also the `KeyExpansion` has to be unrolled and further pipeline registers are needed. However in the stored-key approach, when multiple flows are processed in a pipelined AES block, the `KeyExpansion` and the sub-keys buffer have to be duplicated. The stored key approach is used by [7] and [9], while other AES implementations rely on key-agile approach [5], [8], [10]. In order to guarantee high-performance and maximum cipher flexibility we decided to resort to the key-agile approach.

**Data-Path Placement** – *Data-path placement* involves using the high-level structure of the computation and the data flow to better place the design. This allows many benefits, including shorter wires, more physically compact layout, and faster and easier place-&-route step. The shortening of the wires and their associated delays enables improved performance on FPGA, since a considerable contribution to the critical path is due to the net delay. Unfortunately a post synthesis hand layout on the FPGA of a given block demands for a high effort and time. Instead, we designed and implemented each basic block of AES cipher using appropriate VHDL attributes to address synthesis and placement, then we placed the overall architecture in accordance with the data flow. This approach is much more handy than a hand layout with a florplan editor. Some details about the implementation and the resulting placed data-paths are reported in Section 4 and in Fig. 4, respectively.

**Target Device, Design Flow and Tools** – The FPGA part we chose is a Xilinx Virtex-E 2000-8bg560. The Virtex device family appears to be a good representative for a modern FPGA, and is not fundamentally different from devices from other vendors. The Xilinx Virtex series of FPGAs, consists of Configurable Logic Blocks (CLBs) and interconnection circuitry tiled to form a chip. Each CLB consists of two slices, and each slice contains two 4 inputs Look-Up Tables (LUT), 2 flip-flops (FF), and associated carry chain logic. Each LUT can either be used as a 16x1 bit RAM, or as a 1-16 cycle delay shift register. The Xilinx Virtex-E 2000 presents 19200 slices and 19520 tristate buffers and incorporates also fully synchronous dual-ported 4096*b* block memories named Block SelectRAM. Block SelectRAM memories (BRAM) are organized in columns and each BRAM is four CLBs high. As far as design tools are concerned, we used Aldec Active VHDL 4.2 for simulation, Synplicity Synplify Pro 7.1, Synopsys FPGA Express, Xilinx XST for synthesizing VHDL, and Xilinx ISE 4.1 for place-&-route and timing analysis. In our AES design experience Synplify appeared to achieve better synthesis results with respect to Xilinx XST and FPGA Express.

## 4 AES Blocks Design and Implementation

**Iterative Architecture** – The schematic of the proposed iterative AES block, the *Round* and the *KeyGen* blocks are reported in Fig. 3. The *Round* block can be configured to implement both the `Round` and the `FinalRound` functions of Fig. 1, while the `InitialRound` is instanced as a single block. The AES block (see Fig. 3) presents a latency equal to $N_R + 1$ clock ticks, but the processing of a new plain text can start and a cipher-text is output every $N_R$ ticks. This is allowed by the overlapping of the first serial step of the processing (*InitialRound*) of a plain text block with the last *Round* of the previous plain text processing. We implemented the *InitialRound* (that is a 128*b* xor gate) and the following multiplexer together in the same set of 128 LUTs, since each LUT realizes any 4 inputs function. In other 128 LUTs the 2-to-1 multiplexer and the *AddRoundKey* can be accommodated. The matrix multiplication involved in the *MixColumns* block can be realized as suggested in [2] using the `xtime` function. In fact the *MixColumns* can be realized as a net of xor gate, whereas the maximum fan-in of a xor is 5 and this gate requires 2 LUTs. For implementing the *SubBytes* we used the Xilinx Virtex-E BRAMs. Each BRAM is configured as a dual-port synchronous $256 \times 8b$ words RAM, and it implements 2 s-boxes accessing two locations concurrently. Hence 8 BRAMs suffice for each *SubBytes* blocks. Since the BRAM are synchronous, they can also realize the *Register* before *SubBytes*. Finally the *ShiftRows* can be simply realized by hardwiring. As far as the *KeyGen* block is concerned, the *SubWord* is made up of 4 s-boxes and requires 2 BRAMs which implement also the 32*b* *Register* of Fig. 3. Four 32*b* registers balance the latency due to the 32*b* *Register*. The *Controller* has 4 main states and uses a counter to store the number of round functions applied to the current plain text. The *Controller* runs concurrently in pipelining with the data-path, in a such way that elaboration of data and sequencing of operations can be overlapped. The control signals are buffered into flip-flops that also break up the propagation of

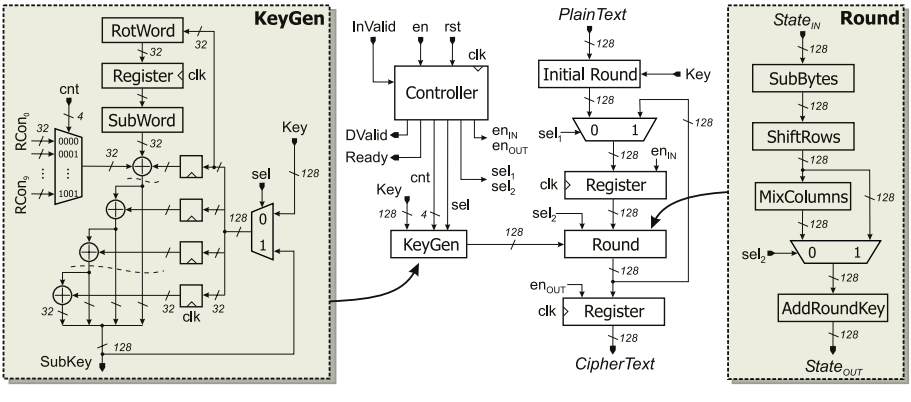


**Fig. 3.** The schematic of the iterative AES architecture.

these signals to the data-path, greatly limiting the contribution of the net delay on the critical path.

As far as the placement of the blocks is concerned, we decided to enclose the data-path of the iterative AES block in a box 4 CLBs high. Actually a *SubBytes* block requires 8 BRAMs and each BRAM is 4 CLBs high. 32 CLBs can accommodate 128 slices and 128 FF-slices and the data-path parallelism is exactly 128*b*. Therefore each block of Fig. 3 is placed in order to occupy a different number of 32 CLB columns, and the blocks are ordered in the same way as they process the data. The *Round* function requires a single column. The *MixColumns* is accomodated in 3 columns, since the `xtime` function is allocated in the first one but it does not fill all the available LUTs, while the remaining 2 columns are completely used for 5-inputs xor-s. The *KeyGen* employees 2 BRAMs and so it is enclosed in a box 8 CLBs high. For the *Controller* we did not impose any placement constraints. The resulting floorplan of the iterative AES block is reported in Fig. 4a. The implementation results are reported in Fig. 5. The same AES block without any optimization and placement constraints requires 1736 slices and presents a $T_{CK}$ of 19,9 ns, and so our optimizations achieve great area saving (-74%) and throughput improvement (+55%).

**5-Pipelined Iterative Architecture** – Interleaving encryption flows allows to use the pipelining technique also for the iterative architecture. We analyzed the delay of each block in the AES core, in order to insert pipeline registers balancing the delay of the stages. We introduced in the schematic of Fig. 3 one 128*b* register between *SubBytes* and the wiring implementing *ShiftRows*, and two registers before the inputs of *AddRoundKey*, to obtain a 5 pipeline stages version of the iterative architecture presented in the previous paragraph. The introduction of the pipeline registers does not remarkably impact on the slices count, since many slices are partially occupied by the LUTs of the iterative non-pipelined AES block. As far as the sub-keys generator is concerned, several

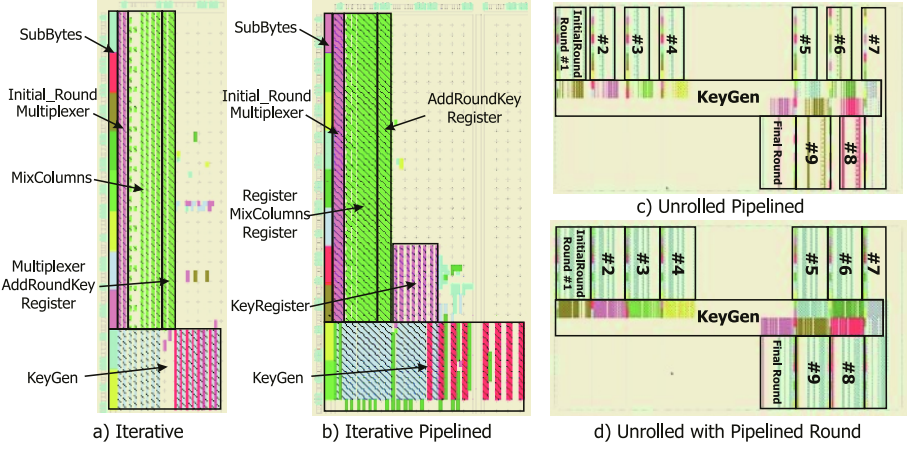


**Fig. 4.** The floorplans of all the proposed implementations of AES.

| Design | Device | Pipeline [stages] | Clock [Mhz] | Th. [Gbps] | Slices (BRAM) | Mbps/ Slices |
|---|---|---|---|---|---|---|
| Weaver et al. [5] | XVE600-8 | 1 | 60 | 0,69 | 460 (10) | 1,5 (0,40) |
| Labbé et al. [7] | XCV1000-4 | 1 | 31 | 0,39 | 2151 (4) | 0,18 (0,15) |
| Amphion CS5220 [10] | XVE-8 | 1 | 101 | 0,29 | 421 (4) | 0,69 (0,31) |
| Amphion CS5230 [10] | XVE-8 | 1 | 91 | 1,06 | 573 (10) | 1,9 (0,57) |
| **Iterative** | XVE2000-8 | 1 | 79 | **1** | **446 (10)** | **2,3 (0,58)** |
| Weaver et al. [5] | XVE600-8 | 5 | **158** | 1,75 | 770 (10) | 2,3 (0,85) |
| Labbé et al. [7] | XCV1000-4 | 5 | 30 | 1,91 | 8767 (4) | 0,22 (0,21) |
| **5-Pipe Iterative** | XVE2000-8 | 5 | 142 | **1,82** | 648 (10) | **2,8 (0,94)** |
| Elbirt et al. [9] | XCV1000-4 | 4 | 40 | 0,98 | 5449 (0) | 0,18 (0,18) |
| Elbirt et al. [9] | XCV1000-4 | 10 | 31 | 1,88 | 10923 (0) | 0,17 (0,17) |
| **1-Pipe Fully Unr.** | XVE2000-8 | 10 | 70 | **8,9** | 2778 (100) | **3,2 (0,57)** |
| **5-Pipe Fully Unr.** | XVE2000-8 | 50 | **158** | **20,3** | 5810 (100) | **3,5 (1,1)** |

**Fig. 5.** Performance results for the proposed, commercial and academical implementations of AES.

changes to the schematic of Fig. 3 are needed to realize a 5-stages pipelined key agile *KeyRegister*. Registers are inserted in the place of the dotted lines in Fig. 3. The registers after the multiplexer become shift-registers to keep aligned the data: the first 32*b* register on the top remains unchanged, the second and the third are delayed of 2 clock ticks, while the last of 3 clock ticks. Also the '1' input of the feedback mux is delayed of other 2 clock ticks. A register (indicated as *KeyRegister* in Fig. 4b) on the output *SubKey* to split the propagation of the sub-keys towards the *Round*. We used LUTs configured as shift registers to realize efficiently these flip-flop chains. Finally the *Controller* has to be slightly modified in order to keep track of the current blocks being processed.

**1-Pipelined and 5-Pipelined Fully Unrolled Architecture** – The 1-pipelined Architecture and the 5-pipelined Architecture implement the completely unrolled AES loop and differ in the level of pipelining. The *1-pipelined Fully Unrolled Architecture*, which uses one pipeline stage per round, is obtained simply juxtaposing ten replica of the *Round* and *KeyGen* depicted in Fig. 3, and using a pipeline register for each round. The first round is an *InitialRound*, while the last round is a *FinalRound* block, and both are simpler than the intermediate *Round*. The *5-pipelined Fully Unrolled Architecture* makes use of 5 levels of pipelining per round, and is obtained from the 1-pipelined Architecture, increasing the pipelining level per round up to 5, following the same remarks described in the design of the 1-pipelined Iterative Architecture. In order to place 10 rounds, 10 BRAM columns are needed, but in a Virtex-E 2000 there are only 8 BRAM columns. Since each BRAM column has 20 BRAMs and each *Round* and *KeyGen* occupy 10 BRAMs, we decided to dispose the first 6 rounds on the upper half of the device, the round 7 split into two parts on the upper and lower half, while the remaining rounds on the lower part. The net delays in the round 7 are relevant, since the basic blocks are scattered on the device, so we inserted an additional pipe register to avoid that this could impact on the critical path. The resulting floorplans of these two AES implementations are reported in Fig. 4c,d and the performance results are given in the last two rows of the Fig. 5.

## 5 Performance Results and Comparison with Related Work

We adopted three different parameters to evaluate proposed implementations of AES: 1) the throughput $T$ considered as the maximum encryption rate; 2) the cost $A$ in terms of area as number of Virtex-E slices and BRAMs; 3) the throughput per area $T/A$, which measures the contribution of each slice to the throughput and hence the efficiency of the implementation. Performance results for each AES block of Section 4, and for other academical and commercial implementations are summarized in Fig. 5. The results are expressed in terms of the pipelining levels, the maximum clock frequency, the throughput, the area (as slices and as BRAMs) and finally $T/A$. It is worth noting that a dual-port $256 \times 8b$ BRAM can be replaced by a distributed memory composed of 256 LUTs (128 slices). Therefore $T/A$ is also reported (in brackets) whereas $A$ is given by the total number of slices using distributed memories and no BRAMs. The reported implementations lie only on 2 types of devices, namely XCV-4 and XVE-8, with differently available resources. These devices are based on the same building block, i.e. a slice containing 2 LUTs and 2 FFs. Our iterative architectures with 1 and 5 pipelining levels have the best $T$ (ranging from 1 to 1,82 Gbps) and $T/A$ among the other iterative implementations. Our fully unrolled architecture with 50 pipeline stages provides the highest encryption throughput (20,3 Gbps), the highest clock frequency (158 MHz) and the best $T/A$, among the reported implementations. These results demonstrate that the use of techniques such as unrolling and pipelining allow to explore the design space tailoring the performance and area requirements. These techniques are also able to deliver very high performance. Furthermore the accurate data-path placement is effective in reducing the area and improving the clock period of the AES blocks. The placed and routed AES blocks with the placement constraints present a clock period improvement ranging between the 36% and the 55%, with respect to the corresponding design without constraints. In [8] an iterative architecture implemented in a chip using a 0,18 micron CMOS technology with core supply at 1,8V is reported. It can deliver 1,6 gigabit per second encryption rate with $128b$ keys, while our best FPGA iterative implementation reaches 1 gigabit per second. So the floorplan aware design, the use of the architectural features of an FPGA family, and the deep pipelining enable a low-cost technology such as FPGA to reach performance comparable to an ASIC implementation, at least for algorithms which can be mapped in a highly regular structure.

## 6 Conclusions

This paper has addressed the design and the FPGA implementation of a fully key agile AES encryption core with 128-bit keys. We discussed the effectiveness of several techniques, such as accurate floorplanning, the unrolling, tiling and pipelining transformations to explore the design space. We dealt with the issues concerning the use of the pipelining in presence of feedback loops and analyzing

several solutions. Based on these solutions, four architectures with different level of parallelism have been designed and implemented on a Virtex-E FPGA part. Performance results show that the presented implementations achieve better performance with respect to other academical and commercial AES block.

## References

1. J. Daemen and V. Rijmen, "AES Proposal: Rijndael, AES Algorithm Submission", September 1999, available at http://www.nist.gov/CryptoToolkit
2. National Institute of Standards and Technology (NIST), FIPS Publication 197, "Advanced Encryption Standard (AES)", November 2001.
3. A. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1996.
4. National Institute of Standards and Technology (NIST), Special Publication 800-38A, "Recommendation for Block Cipher Modes of Operation", December 2001.
5. N. Weaver, J. Wawrzynek, "Very High Performance, Compact AES Implementation in Xilinx FPGAs", September 2002, available at http://www.cs.berkeley.edu/~nweaver/sfra/rijndael.pdf
6. C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming", Third Caltech Conference On VLSI, March 1993.
7. A. Labbé, A. Pérez, "AES Implementations on FPGA: Time-Flexibility Tradeoff", Proceedings of FPL02, pp. 836-844.
8. P.R. Schaumont, H. Kuo, and I.M. Verbauwhede, "Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor", Proceedings of the International Design Automation Conference 2002 (DAC02), pp. 634-639.
9. A.J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists", IEEE Trans. on VLSI Systems, Vol.9, No.4, August 2001, pp. 545-557.
10. Amphion Semiconductor, "CS5210-40: High Performance AES Encryption Cores", Amphion(TM), Febrauary 2003, available at http://www.amphion.com/cs5210.html