# DSPs, BRAMs and a Pinch of Logic: New Recipes for AES on FPGAs

Saar Drimer[1], Tim Güneysu[2], Christof Paar[2]
[1]Computer Laboratory, University of Cambridge, UK
[2]Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
http://www.cl.cam.ac.uk/~sd410, {gueneysu, cpaar}@crypto.rub.de

## Abstract

*We present an AES cipher implementation that is based on the BlockRAM and DSP units embedded within Xilinx's Virtex-5 FPGAs. An iterative "basic" module outputs a 32 bit column of an AES round each clock cycle, with a throughput of 1.76 Gbit/s when processing two 128 bit inputs. This construct is replicated four times for a 128 bit datapath for a full AES round with 6.21 Gbit/s throughput when processing eight inputs. Finally, the "round" module is replicated ten times for a fully unrolled design that yields over 55 Gbit/s of throughput. The combination and arrangement of the specialized embedded functions available in the FPGA allows us to implement our designs using very few traditional user logic elements such as flip-flops and lookup tables, yet still achieve these high throughputs. The complete source code for these designs is made publicly available for use in further research and for replicating our results. Our contribution ends with a discussion of comparing cipher implementations in the literature, and why these comparisons can be meaningless without a common reporting style, platform, or within the context of a specific constrained application.*

## 1 Introduction

The Advanced Encryption Standard (AES) [16] is a block cipher used in many applications with a rich literature discussing how to optimize implementations of it for both software and hardware. Most available AES implementations for reconfigurable hardware, however, are based on traditional configurable logic and do not exploit the full potential of modern devices. Our implementation focuses on new embedded functions inside of the Xilinx Virtex-5 FPGA [17], such as large dual-ported RAMs and digital signal processing (DSP) blocks [18] with the goal of minimizing the use of registers and look-up tables that could otherwise be used for other functions. Therefore, the design we present will be especially appealing in applications where user logic is scarce, yet not all embedded memory and DSP blocks are used. We have created a "basic" eight-stage pipeline module based on a combination of two 36 Kbit BlockRAM (BRAM) and four DSP blocks, which outputs one 32 bit column of an AES round each cycle with a feedback loop for iterative operation. This basic module is replicated four times for a full AES round with a 128 bit datapath, which, in turn, is replicated ten times for a fully unrolled operation; we do not include the key expansion function in these designs. All Verilog source code for the three variants is publicly available for reuse, evaluation, and reproduction of our results.

We begin with a background discussion in Sections 2 and 3 followed by the implementation details in Section 4 and results in Section 5. In Section 6 we discuss comparing results of cipher implementations from multiple sources, along with a brief survey of previous work.

## 2 AES cipher operation

We encrypt data when we want to keep it confidential and illegible to people who are not meant to see it in its "plaintext" form. Encryption is used in a wide range of applications, some requiring large amounts of data to be encrypted or decrypted at very high speeds. Symmetric encryption using block ciphers is often used, with the security relying on a pre-established secret key shared between sender(s) and receiver(s). AES has been designed as a substitution-permutation network (SPN) and uses between 10 to 14 encryption rounds (depending on the length of the key) for a full encryption and decryption of one 128 bit block. In a single round, the AES operates on all of the 128 input bits represented as a $4 \times 4$ matrix of bytes. Fundamental operations of the AES are performed based on byte-level field arithmetic over the Galois Field $GF(2^8)$ so that operands can be represented in 8 bit vectors. The AES cipher has been designed to be efficient in both hardware and software, and is versatile in that it can be made either area-optimized, iterative, and slow, or "unrolled" and fast by parallelizing round operations and pipelining. Its

8 bit vector representation allows implementations on very small processing units, while 128 data paths allow for gigabit throughput. We now describe the operation of AES and how we use it in our designs.

When describing the cipher's operations, $A$ is denoted as the input block consisting of bytes $a_{i,j}$ in columns $C_j$ and rows $R_i$, where $j$ and $i$ are the respective indices ranging between 0 and 3 .

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

An AES round consists of four basic operations on $A$:

1. *SubBytes*: all input bytes of $A$ are substituted with values from a non-linear $8 \times 8$ bit S-Box.
2. *ShiftRows*: the bytes of rows $R_i$ are cyclically shifted to the left by 0, 1, 2 or 3 positions.
3. *MixColumns*: columns $C_j$ are matrix-vector-multiplied by a matrix of constants in $GF(2^8)$.
4. *AddRoundKey*: a round key $K_i$ is added to the input using $GF(2^8)$ arithmetic.

The sequence of these four operations define an AES round, and they are iteratively applied for a full encryption or decryption of a single 128 bit input block. Since some of the operations above rely on $GF(2^8)$ arithmetic we are able to combine them into a single complex operation. The Advanced Encryption Standard defines such an approach for software implementations on 32 bit processors with the use of large lookup tables. This approach requires four 8 to 32 bit lookup tables for the four round transformations, each the size of 8 Kbit. We can compute these transformation tables, $T_{k[0..3]}$, in the following way:

$$T_0[x] = \begin{bmatrix} S[x] \times 02 \\ S[x] \\ S[x] \\ S[x] \times 03 \end{bmatrix} \qquad T_1[x] = \begin{bmatrix} S[x] \times 03 \\ S[x] \times 02 \\ S[x] \\ S[x] \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} S[x] \\ S[x] \times 03 \\ S[x] \times 02 \\ S[x] \end{bmatrix} \qquad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \times 03 \\ S[x] \times 02 \end{bmatrix}$$

$S[x]$ denotes a table lookup in the original $8 \times 8$ bit AES S-Box (for a more detailed description of this AES optimization see NIST's FIPS-197 [16]). The last round, however, is unique in that it omits the MixColumns operation, so we need to give it special consideration. There are two ways for computing the last round, either by "reversing" the MixColumns operation from the output of a regular round by another multiplication in $GF(2^8)$, or creating dedicated "last round" T-tables, one per regular T-table. The latter approach will allow us to maintain the same datapath for all rounds, so we chose this method and denote these T-tables as $T_{[j]'}$. With all T-tables at hand, we can redefine all transformation steps of a single AES round as

$$\begin{aligned} E_j^{'} \;=\; & K_{r[j]} \oplus T_0[a_{0,j}] \oplus T_1[a_{1,(j+1 \bmod 4)}] \oplus \\ & T_2[a_{2,(j+2 \bmod 4)}] \oplus T_3[a_{3,(j+3 \bmod 4)}] \end{aligned} \quad (1)$$

where $K_{r[j]}$ is a corresponding 32 bit "sub-key" and $E_j^{'}$ denotes one of four encrypted output *columns* of a full round. We now see that based on only four T-table lookups and four XOR operations, a 32 bit column $E_j^{'}$ can be computed. To obtain the result of a full round, Equation (1) must be performed four times with all 16 bytes.

Input data to an AES encryption can be defined as four 32 bit column vectors $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ with the output similarly formatted in column vectors. According to Equation 1, these input column vectors need to be split into individual bytes since all bytes are required for the computation steps for different $E_j^{'}$. For example, for column $C_0 = (a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0})$ the first byte $a_{0,0}$ is part of the computation of $E_0^{'}$, the second byte $a_{1,0}$ is used in $E_3^{'}$, etc. Since fixed (and thus simple) data paths are preferable in hardware implementations, we have rearranged the operands of the equation to align the bytes according to the input columns $C_j$ when feeding them to the T-table lookup. In this way, we can implement a unified data path for computing all four $E_j^{'}$ for a full AES round. Thus, Equation (1) transforms into

$$\begin{aligned} E_0^{'} &= K_{r[0]} \oplus T_0(a_{0,0}) \oplus T_1(a_{1,1}) \oplus T_2(a_{2,2}) \oplus T_3(a_{3,3}) \\ &= (a_{0,0}^{'}, a_{1,0}^{'}, a_{2,0}^{'}, a_{3,0}^{'}) \\ E_1^{'} &= K_{r[1]} \oplus T_3(a_{3,0}) \oplus T_0(a_{0,1}) \oplus T_1(a_{1,2}) \oplus T_2(a_{2,3}) \\ &= (a_{0,1}^{'}, a_{1,1}^{'}, a_{2,1}^{'}, a_{3,1}^{'}) \\ E_2^{'} &= K_{r[2]} \oplus T_2(a_{2,0}) \oplus T_3(a_{3,1}) \oplus T_0(a_{0,2}) \oplus T_1(a_{1,3}) \\ &= (a_{0,2}^{'}, a_{1,2}^{'}, a_{2,2}^{'}, a_{3,2}^{'}) \\ E_3^{'} &= K_{r[3]} \oplus T_1(a_{1,0}) \oplus T_2(a_{2,1}) \oplus T_3(a_{3,2}) \oplus T_0(a_{0,3}) \\ &= (a_{0,3}^{'}, a_{1,3}^{'}, a_{2,3}^{'}, a_{3,3}^{'}) \end{aligned}$$

where $a_{i,j}$ denotes an input byte, and $a_{i,j}^{'}$ the corresponding output byte after the round transformation. However, the unified input datapath still requires a look-up to all of the four T-tables for the second operand of each XOR operation. For example, the XOR component at the first position of the sequential operations $E_0^{'}$ to $E_3^{'}$ and thus requires the lookups $T_0(a_{0,0}), T_3(a_{3,0}), T_2(a_{2,0})$ and $T_1(a_{1,0})$ (in this order) and the corresponding round key $K_{r[j]}$. Though operations are aligned for the same input column now, it becomes apparent that the bytes of the input column are not processed in canonical order, i.e., bytes need to be swapped for each column $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ first before being fed as input to the next AES round. The required byte transposition is reflected in the following equations:

$$\begin{aligned} C_0 &= (a'_{0,0}, a'_{3,0}, a'_{2,0}, a'_{1,0}) \\ C_1 &= (a'_{1,1}, a'_{0,1}, a'_{3,1}, a'_{2,1}) \\ C_2 &= (a'_{2,2}, a'_{1,2}, a'_{0,2}, a'_{3,2}) \\ C_3 &= (a'_{3,3}, a'_{2,3}, a'_{1,3}, a'_{0,3}) \end{aligned} \tag{2}$$

Note that the given transpositions are static so that they can be efficiently hardwired in our implementation.

The AES also has a "key schedule" or "key expansion" operation, which takes the input key and derives from it "sub-keys" for each round, denoted here as $K_{r[j]}$. Many applications, however, use a single key for processing a bulk of data, so the round keys need only to be computed at the initialization phase and then remain static until a new key is required. In this paper we do not include the key expansion function as part of the implementation mainly because of the above reasoning, but also because there already exist efficient designs for this function; we will, however, propose how this function can be integrated into our design in Section 5.1. One final operation has not been mentioned yet and lies between the key schedule and the round operation; this is an XOR operation of the input key and the input 128 bit block. We omit this operation when reporting our results, though we discuss how it can be performed at a minor expense of performance and resources.

### 2.1 Decryption

AES encryption and decryption of data require different treatment so that usually separate hardware components and a significant logic overhead becomes necessary to support both operations. With our approach, all primitive operations are encoded into T-tables for encryption, so that we can apply a similar strategy for decryption by creating tables representing the inverse cipher transformation. Hence, we can basically support an encryptor and decryptor engine with the same circuit by only swapping the values of the transformation tables. As with Equation (1), decryption of columns $D'_j$ is governed by the following set of equations:

$$\begin{aligned} D'_0 &= K_{r[0]} \oplus I_0(a_{0,0}) \oplus I_1(a_{1,3}) \oplus I_2(a_{2,2}) \oplus I_3(a_{3,1}) \\ &= (a'_{0,0}, a'_{1,0}, a'_{2,0}, a'_{3,0}) \\ D'_3 &= K_{r[3]} \oplus I_3(a_{3,0}) \oplus I_0(a_{0,3}) \oplus I_1(a_{1,2}) \oplus I_2(a_{2,1}) \\ &= (a'_{0,3}, a'_{1,3}, a'_{2,3}, a'_{3,3}) \\ D'_2 &= K_{r[2]} \oplus I_2(a_{2,0}) \oplus I_3(a_{3,3}) \oplus I_0(a_{0,2}) \oplus I_1(a_{1,1}) \\ &= (a'_{0,2}, a'_{1,2}, a'_{2,2}, a'_{3,2}) \\ D'_1 &= K_{r[1]} \oplus I_1(a_{1,0}) \oplus I_2(a_{2,3}) \oplus I_3(a_{3,2}) \oplus I_0(a_{0,1}) \\ &= (a'_{0,1}, a'_{1,1}, a'_{2,1}, a'_{3,1}) \end{aligned}$$

This requires the following inversion tables (I-Tables), where $S^{-1}$ denotes the inverse $8 \times 8$ S-Box for the AES decryption:

$$I_0[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{0E} \\ S^{-1}[x] \times \texttt{09} \\ S^{-1}[x] \times \texttt{0D} \\ S^{-1}[x] \times \texttt{0B} \end{bmatrix} \qquad I_1[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{0B} \\ S^{-1}[x] \times \texttt{0E} \\ S^{-1}[x] \times \texttt{09} \\ S^{-1}[x] \times \texttt{0D} \end{bmatrix}$$

$$I_2[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{0D} \\ S^{-1}[x] \times \texttt{0B} \\ S^{-1}[x] \times \texttt{0E} \\ S^{-1}[x] \times \texttt{09} \end{bmatrix} \qquad I_3[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{09} \\ S^{-1}[x] \times \texttt{0D} \\ S^{-1}[x] \times \texttt{0B} \\ S^{-1}[x] \times \texttt{0E} \end{bmatrix}$$

We can see that compared to encryption, the input to the decryption equations is different at two positions for each decrypted column $D'_j$. But, instead of changing the datapath from the encryption function, we can change the order in which the columns $D'_j$ are computed so that instead of computing $E'_0, E'_1, E'_2, E'_3$ for encryption, we determine the decryption output in the column sequence $D'_0, D'_3, D'_2, D'_1$. Preserving the datapath by only changing the content of the tables will allow us to use (nearly) the same circuit for both functions, as we shall see in Section 4.

## 3 Embedded FPGA elements

Since the early 2000s, FPGA vendors have started designing into the FPGA popular functions that previously required separate peripheral devices. Examples are large memory blocks, clock managers, hard microprocessors, and fast serial transceivers. We are also seeing a process of industry-specific sub-families within a single family of devices that cater to embedded, DSP, military, and automotive applications; this means that the distribution of the various embedded blocks is different across family members. Of particular interest to us is the integration of large memory elements and arithmetic hard cores for efficient multiplication and addition operations with low carry propagation times. Since the U.S. NIST adopted the Rijndael cipher as the AES in 2001, it has been implemented in various ways on both FPGAs and ASICs. Early AES designs were usually straightforward implementations of loop unrolled or pipelined architectures, mostly on FPGAs utilizing a vast amount of user logic elements [4, 9]. Particularly, the required $8 \times 8$ S-Boxes of the AES have been implemented in the lookup tables (LUT) of the user logic usually requiring large portions of the device. More advanced approaches [11, 15, 6, 2] used the on-chip memory components of FPGAs, implementing the S-Box tables in separate RAM sections on the device. Since RAM capacities were limited in previous generations of FPGAs, the majority of implementations only mapped the $8 \times 8$ S-Box into

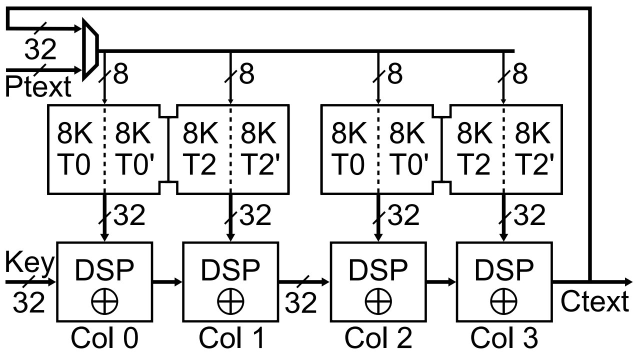

**Figure 1. The basic construct structure. Each dual ported BRAM contains four T-tables, two for the first nine rounds, and two for the last one. Each DSP48E block performs a 32 bit bitwise XOR operation. After passing through the four DSP blocks, column results are fed back as the input to the next round.**

the memory while all other AES operations like ShiftRows, MixColumns and the AddRoundKey are realized using traditional user logic, and proved costly in terms of flip flops and LUTs. To our knowledge, only two previous implementations [5, 2] have transferred the software architecture based on the T-table to FPGAs for saving logic cells required for the AES primitive operations. However, due to the large tables and the restricted memory capacities on those devices, certain functionality must be still encoded in user logic. Our contribution is the first T-table-based AES-implementation that efficiently used device-specific features available in one particular FPGA family.

## 4 Implementation

In the previous section we have introduced the lookup-table method for implementing the AES round optimized for 32 bit microprocessors. Now we will demonstrate how to adapt this software-oriented approach into modern reconfigurable hardware devices in order to achieve high throughput for modest amounts of resources. We use the Xilinx Virtex-5 FPGA which has advanced features that are useful for our application beyond traditional LUTs and registers. These are dual ported 36 Kbit BlockRAMs (BRAM) — ones that have independent address and data buses for the same stored content — and versatile digital signal processing (DSP) cores. The DSP cores allow the designer to implement timing- or resource-critical functions such as arithmetic operations on integers or Boolean expressions that would otherwise be considerably slower or resource demanding if implemented with “ordinary” logic elements. The DSP blocks were introduced in the Virtex-4 family of FPGAs to perform $18\times18$ bit integer along with a 48 bit accumulator, though they were limited to 24 bit bit-wise logic operations. 48 bit bit-wise logic operations were added in Virtex-5, and can run at up to 550 MHz, the maximum frequency rating of the device. The internal datapath inside of the DSP block is 48 bit wide, except for integer multiplication. The Virtex-5 DSP blocks come in pairs that span the height of five configurable logic blocks (CLB), and they can be efficiently cascaded between pairs with an additional dedicated paths to adjacent DSP tiles. A single dual-ported 36 Kbit BRAM also spans the height of five CLBs and matches the height of the pair of DSP blocks, with a fast datapath between them. Our initial observation was that the 8 to 32 bit lookup followed by a 32 bit XOR AES operation perfectly matched this architectural alignment for efficient and fast implementation. Based on these primitives, we developed a basic AES module that performs a quarter (one column) of an AES round transformation given by Equation (1). We have designed it so that it allows efficient placing and routing of components such that it can operate at the maximum device frequency of 550 MHz. Furthermore, our basic module is designed such that it can be replicated for higher throughput.

### 4.1 Basic module

The basic construct we started out with is shown in Figure 1. Since each column requires all four T-table lookups with their last-round T-table counterparts, that means that we needed to fit a total of eight 8 Kbit T-tables in a single 36 Kbit dual-port RAM. As we discussed in Section 2, for performance and resource efficiency reasons we opted against “reversing” the last operation and searched for a solution that would enable us to fit all tables into a single BRAM. We realized that our design can use the fact all T-tables are byte-wise transpositions of each other, such that by cyclically byte-shifting of the BRAM’s output for T-table $T_0$ we can produce the output of $T_1$, $T_2$ and $T_3$. Using this observation, we only store $T_0$ and $T_2$, and also their last-round counterparts $T_{0'}$ and $T_{2'}$ in a single BRAM. Using a single byte circular right rotation $(a,b,c,d) \rightarrow (d,a,b,c)$, $T_0$ becomes $T_1$, and $T_2$ becomes $T_3$ and the same for the last round’s T-tables. In hardware, this requires a 32 bit 2:1 multiplexer at the output of each BRAM with a select signal from the control logic. For the last round, a control bit is connected to a high order address bit of the BRAM to switch from the regular T-table to the last round’s T-table. The memory layout is shown in Figure 1: the first 8 bits of the address is the input byte $a_{i,j}$ to the transformation, bit 9 controls the choice between regular and last round T-table, while bit 10 chooses between $T_0$ and $T_2$. Thus, a dual-port 32 Kbit BRAM with three control bits, and a 2:1 32 bit mux allows us to output all the needed T-tables for two columns.

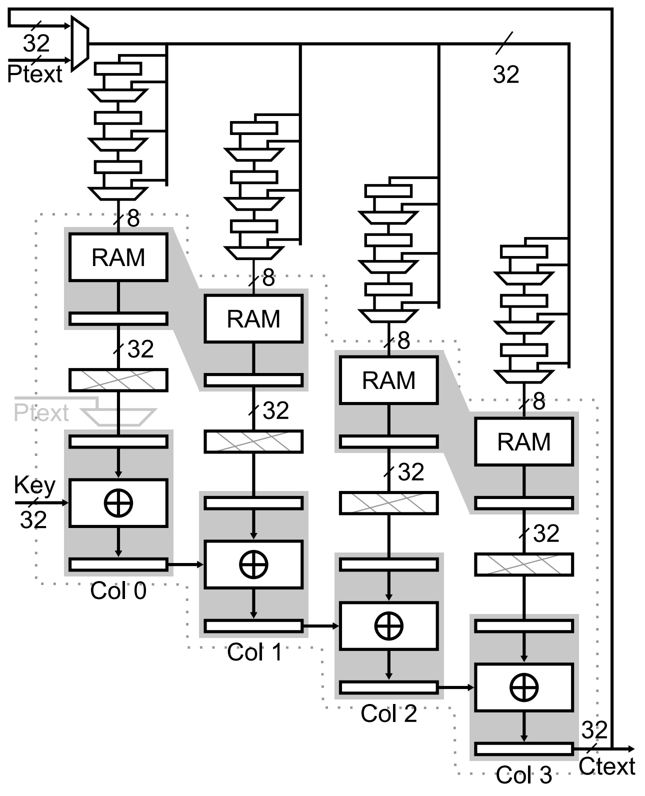

**Figure 2. The "basic" iterative round (without control logic). Plaintext $P_{text}$ is chosen as the initial input, then output data is fed back through 8 bit shift registers for a complete AES encryption. The pipeline stage between BRAMs and DSPs block is used as an optional 8 bit right-shift when $T0$ and $T2$ are turned into $T1$ and $T3$, respectively.**

Using two such BRAMs with identical content, we get the necessary lookups for four columns, each capable of performing all four T-table lookups.

Both the BRAMs and DSP blocks provide internal input and output registers for pipelining along the datapath; we get these registers "for free" without use of any flip flops in the fabric. At this point, we already had six pipeline stages that could not have been easily removed if our goal was high throughput. So we decided that instead of trying to reduce pipeline stages, we could *add* two more so that we are able to process two input blocks at the same time, doubling the throughput. One of these added stages is the 32 bit register after the 2:1 multiplexer that shifts the T-tables at the output of the BRAM; these are the only user logic registers we use for the basic construct (shown inside dotted line in Figure 2).

A full AES operation is implemented by operating the basic construct with an added feedback scheduling in the datapath. Combined with BRAM lookups, we assigned a cascade of DSP blocks to perform the four XOR operations required for computing the AES column output according to Equation (1). For feeding in the corresponding $a_{i,j}$ for the lookup into the BRAM, we added a sequence of three 8 bit loadable shift registers and an input multiplexer for each column. These 24 bit registers are loaded in sequence, the leftmost ($C_0$) on the first cycle, and the one to its right on the next, and so on.

This construct has eight pipeline stages with the following operations, in order: lookup $L$, where the 8 to 32 bit T-table lookup is performed within the BRAM; register $R$ is the BRAM's output register; transform $T_{[0..3]}$, where $T_0$ and $T_2$ are optionally shifted into $T_1$ and $T_3$ content, respectively; DSP $D$ input register; and $\oplus$, the exclusive-or operation. There are also four columns ($C_{[0..3]}$) which are staggered as shown in Figure 2. As previously mentioned, the shaded pipeline stages are part of the BRAM or DSP blocks, not "traditional" user logic in the form of CLB flip-flops — our goal was to minimize the use of these resources.

Table 1 shows the eight pipeline stages in the first thirteen clock cycles; the plaintext at the top is fed in four 32 bit words, one word per cycle. The first column output $E_0^{'}$ is produced on the $8^{th}$ clock cycle and is fed back to the input for processing the second round. Notice that the corresponding outputs are produced as defined by Equation 1. For the second round, after eight clock cycles, the control logic chooses the feedback rather than the plaintext input using a 32 bit 2:1 mux. Our decision to add two pipeline stages to interleave two plaintexts is apparent Table 1, as we can see that each pipeline stage is performing an operation only four out of every eight cycles. This allows us to feed two consecutive 128 bit blocks one after another, in effect doubling our throughout without any additional complexity.

A grayed-out multiplexer is shown in Figure 2 as an alternative input, which makes it easy to perform the XOR operation of the key and initial input prior to the first round. For four consecutive clock cycles, $C_0$'s DSP performs the XOR operation while the output passes through the other DSPs; this results in the initial round input appearing at the top of the pipeline and the sequence continues as previously described. We have implemented this design, and noticed an expected slight degradation in performance due to the insertion of a 32 bit 2:1 mux in the datapath. There are also additional signals required for the control logic, but those do not affect performance. Finally, we also tried a different approach for computing columns using the same basic structure, but instead of feeding the output of each DSP to the one on its right, the data is fed back onto itself for the next XOR operation with the data arriving from the

|  | Cycle | Key | C0 | C1 | C2 | C3 | Out |
|---|---|---|---|---|---|---|---|
|  | 13 |  | – | $a'_{12}$ | $a'_{23}$ | $a'_{32}$ |  |
|  | 12 |  | $a'_{01}$ | $a'_{10}$ | $a'_{21}$ | $a'_{33}$ |  |
|  | 11 |  | $a'_{02}$ | $a'_{13}$ | $a'_{22}$ | – |  |
|  | 10 |  | $a'_{03}$ | $a'_{11}$ | – | – |  |
| I | 9 |  | $a'_{00}$ | – | – | – |  |
| N | 8 |  | – | – | – | – |  |
| P | 7 |  | – | – | – | $a_{30}$ |  |
| U | 6 |  | – | – | $a_{23}$ | $a_{31}$ |  |
| T | 5 |  | – | $a_{12}$ | $a_{20}$ | $a_{32}$ |  |
|  | 4 |  | $a_{01}$ | $a_{10}$ | $a_{21}$ | $a_{33}$ |  |
|  | 3 |  | $a_{02}$ | $a_{13}$ | $a_{22}$ | – |  |
|  | 2 |  | $a_{03}$ | $a_{11}$ | – | – |  |
|  | 1 |  | $a_{00}$ | – | – | – |  |
|  | 1 |  | $L$ |  |  |  |  |
|  | 2 |  | $R$ | $L$ |  |  |  |
| P | 3 |  | $T$ | $R$ | $L$ |  |  |
| I | 4 |  | $D$ | $T$ | $R$ | $L$ |  |
| P | 5 | $K_{0[0]}$ | $\oplus\searrow$ | $D$ | $T$ | $R$ |  |
| E | 6 | $K_{0[1]}$ | $\oplus\searrow$ | $\oplus\searrow$ | $D$ | $T$ |  |
| L | 7 | $K_{0[2]}$ | $\oplus\searrow$ | $\oplus\searrow$ | $\oplus\searrow$ | $D$ |  |
| I | 8 | $K_{0[3]}$ | $\oplus\searrow$ | $\oplus\searrow$ | $\oplus\searrow$ | $\oplus\rightarrow$ | $E'_0$ |
| N | 9 |  | $L$ | $\oplus\searrow$ | $\oplus\searrow$ | $\oplus\rightarrow$ | $E'_1$ |
| E | 10 |  | $R$ | $L$ | $\oplus\searrow$ | $\oplus\rightarrow$ | $E'_2$ |
|  | 11 |  | $T$ | $R$ | $L$ | $\oplus\rightarrow$ | $E'_3$ |
|  | 12 |  | $D$ | $T$ | $R$ | $L$ |  |
|  | 13 | $K_{1[0]}$ | $\oplus\searrow$ | $D$ | $T$ | $R$ |  |

**Table 1. Initial 13 clock cycles of the eight pipeline stages computing a plaintext input. The stages are: RAM lookup $L$; RAM output register $R$; transform $T$; DSP input register $D$; and, DSP XOR $\oplus$. After eight cycles the output column $E'_0$ is used as input to the next round, and so on.**

BRAM. We found, however, that this requires the input of a key to each DSP block, extra control logic, different operating modes for the DSP , and a 32 bit 4:1 mux to choose between the output of each DSP for the feedback loop. All those introduced extra delays when routed, and performed worse than the original design.

Up to now we focused on the encryption process, though decryption is quite simply achieved with minor modifications to the circuit. As the T-tables are different for encryption and decryption, storing them all would require double the amount of storage, which we want to avoid. Recall, however, that any $T_i$ can be converted into $T_j$ simply by shifting the appropriate amount of bytes. The modification to the design is therefore, replacing the 32 bit 2:1 mux at the output of the BRAM with a 4:1 mux such that all possible byte shifting is possible, and loading the BRAMs with $T_i^E$, $T_{i'}^E$, $T_i^D$ and $T_{i'}^D$, $T^E$ and $T^D$ denoting encryption and decryption T-tables, respectively. Of course, this would degrade the performance because the datapath between BRAM and DSP are now longer. An alternative is to dynamically reconfigure the content of the BRAMs with the decryption T-tables; this can be done from an external source, or even from within the FPGA using the internal configuration access port (ICAP) [17] with a storage BRAM for reloading content through the T-table BRAMs' data input port.

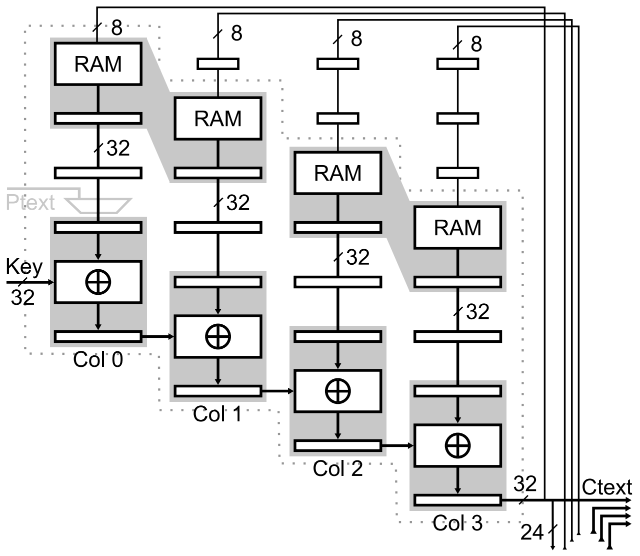

**Figure 3. This construct is replicated four times for a full round (the multiplexer for the initial input is not shown). Except for the input to $C_0$, each column receives the input from the other three instances. The T-tables are static so the shifting of the BRAMs' outputs is fixed by routing.**

## 4.2 Round and unrolled modules

Since the single AES round requires the computation of four 32 bit columns, we can replicate the basic construct four times and add 8, 16, and 24 bit registers at the inputs of the columns. The first of four instances is shown in Figure 3; one byte is fed back to the same instance while 3 bytes are distributed to the other three instances. The latency of this construct is still 80 clock cycles as before, but allows us to interleave eight 128 bit inputs at any given time. This is possible because of the eight pipeline stages, where each of the four instances receives a 32 bit input every clock cycle. As apposed to the previous module, though, the byte arrangements allow that the T-tables be static so the 32 bit 2:1 multiplexers are no longer required. This simplifies the data paths between the BRAMs and DSPs since the shifting can be fixed in routing. The control logic is simple as well, comprising of a 3 bit counter and a 1 bit control signal for choosing the last round's T-tables.

Finally, the natural thing to do was to implement a fully unrolled AES design for achieving maximum throughput. For this, we connected ten instances of the "round" design presented above for an 80-stage pipeline using 80 BRAMs and 160 DSP blocks. Since this design does not require any dynamic control logic it produces a 128 bit output every clock cycle. The initial XOR of the input block with the main key can be done by adding one to four DSP blocks (amount will affect latency) as a pre-stage to the round operation, or be performed in "regular" logic. We now move on to performance results.

## 5 Results

Our Verilog designs target a Virtex-5 LX30 and SX95T devices at their fastest speed grade (-3) using Xilinx Synthesis Technology (XST) and the ISE 9.2i.03 implementation flow. For simulation we used Mentor's ModelSim 6.2g for both behavioral and post place-and-route stages; we did not, however, verify the design on an actual device. Simulation was performed under nominal conditions of voltage (0.95 V) and temperature (85°C) using minimum delay data (netgen option "-s min"). Various effort settings for "map" and "par" were used, along with "multi-pass place-and-route" to achieve best results, which are reported here. In addition, the IOs were ignored for timing ("TIG" constraint), as we consider the cores as a stand-alone function. Once within the context of an application, the designer will have to make sure signals arrive on time at the module's input; we discuss how this will affect performance in Section 6.

The basic AES module as shown in Figure 2 passed timing (post place-and-route) for a frequency just over 550 MHz, the maximum frequency rating of the device. The design requires the following resources: 247 flip-flops, 96 $(8 \cdot 3 \cdot 4)$ for the input shift registers plus 128 $(4 \cdot 32)$ for the pipeline stages in between the BRAMs and DSPs, with the rest used for control logic; 275 look-up tables, mostly functioning as multiplexers; and finally, two 36 Kbit dual-port BRAM (32 Kbit used in each) and four DSP48E blocks. We calculate throughput as follows: given that there are 80 processing cycles operating at 550 MHz and we maintain state of 256 bits in the pipeline stages, we achieve $550 \cdot 10^6 \cdot 256/80 = 1.76$ Gbit/s of throughput. This assumes that the pipeline stages are always full, meaning that the module is processing two 128 bit inputs at any given time; if only one input is processed, the throughput is halved. As we have mentioned, the eight pipeline stages were implemented for the purpose of interleaving two inputs, though the designer can remove pipeline stages to reduce resources. Removing pipeline stages reduces latency, though it may also reduce the maximum frequency, so there is a trade-off that needs to be assessed according to the application.

In the "round" module the basic construct is used four times for a 128 bit-width interface. The maximum frequency reported by the tools post place-and-route was over 485 MHz, and it uses 621 flip-flops, 204 look-up tables, 8 36 Kbit BRAMs (32 Kbit used in each), and 16 DSP48E blocks. Notice that we expect at least $4 \cdot 48 + 4 \cdot 128 = 704$ registers but the tools report only 621. This is because the synthesizer tries to achieve a balanced FF-LUT ratio for better packing into slices so the 2- and 3-stage input shift registers for each basic cells are implemented in eight LUTs each. The latency of 80 clock cycles is the same as the previous design, though now we can maintain state of $128 \cdot 8 = 1024$ bits, thus giving us a throughput of $485 \cdot 10^6 \cdot 8 \cdot 128/80 = 6.21$ Gbit/s when processing eight input blocks. We can see that the complexity of this design reduces the maximum frequency and throughput, though hand placement of DSPs and BRAMs, along with matching the bit ordering to the routing can improve on this performance. As with the basic module, pipeline stages can be removed to minimize the use of logic resources if they are required for other functions and the highest throughput is not required.

Finally, the "unrolled" implementation produces 128 bits of output every clock cycle once the initial latency is complete. We have experimented with eliminating the pipeline stage between the BRAM and DSP to see if it adversely affects performance; this will save us 5,120 registers. We found that the performance degradation is minimal, with the added benefit of having an initial latency of only 70 clock cycles instead of 80. The resulting throughput is $430 \cdot 10^6 \cdot 128 = 55$ Gbit/s. This design operates at a maximum frequency of over 430 MHz and uses 992 flip-flops, 672 look-up tables, 80 36 Kbit BRAMs (only 16 Kbit in each for dec/enc or 32 Kbit for both), and 160 DSP48E blocks; the same balancing act of FF-LUT ratio by the synthesizer occurs here as well. There are very few flip-flops and LUTs compared to what is available in the large SX95T device: 1.68% and 1.14%, respectively, though we use 32% of BRAMs and 25% of DSP48Es.

Out results are summarized in Table 2. Verilog source code for all three modules, including XFLOW commands to replicate the above results, are available at this URL: `http://www.cl.cam.ac.uk/~sd410/aes`

## 5.1 Extensions

We have shown three pipelined architectures for AES operations supporting simultaneous encryption of 2, 8, and 80 128 bit input blocks in electronic codebook (ECB) mode. Cipher block chaining (CBC) mode can be efficiently implemented by adding a further XOR component at the input of the respective AES design to support encryption *or* authentication of up to 80 independent data streams. Additionally, the architectures can be extended to provide authenticated encryption as well, for example, the counter with CBC-MAC (CCM) mode [3] requires two AES operations to be performed in parallel — one for encrypting or decrypting data and another for creating or verifying a message authentication code (MAC), a cryptographic checksum over the data. Thus, with an additional encryption counter, we can easily adapt our modules to provide CCM authenticated encryption for 1, 4, or 40 individual streams of data using the three designs.

In this contribution, we focused on implementations of the AES round operation. Thus, the round key expansion which is mostly done in a pre-computation phase prior encryption has not been considered. In case that the high throughput of our architecture is not required but the key schedule needs to be precomputed on chip without adversely increasing logic resource utilization, our basic AES module can be modified to support the key generation. For the key schedule, the plain $8 \times 8$ S-Boxes and a small set of round constants are required which are XORed with other previously computed 32 bit round keys. Remember that we already store T-tables $T_{[0..3]'}$ for the last round in the BRAMs without the MixColumns operation so that the values of these tables are basically a byte-rotated 8 bit S-Box value. These values are perfectly suited for generating a 32 bit round key from S-Box lookups. Furthermore, our datapath has been specifically designed for 32 bit XOR operations based on the DSP unit. Hence, with appropriate input multiplexers, control logic and a separate BRAM as key store, we can integrate a key schedule in our existing design without introducing much overhead. Alternatively, a separate circuit for the key schedule can be implemented to preserve the regularity of the basic module and the option to operate the design at maximum device frequency. For a minimal footprint, we propose to add another dual-ported BRAM to the design used for storing the expanded 32 bit subkeys (e.g., 44 words for AES-128), the round constants and S-Box entries with 8 bit each. One port of that BRAM is 32 bit wide and feeding the subkeys to the AES module, the other is configured for 8 bit data I/O. With an 8-bit multiplexer, register and XOR connected to the second BRAM port, a minimal and byte-oriented key schedule can be implemented computing the full key expansion in 520 clock cycles.

## 6 Performance evaluations and prior work

Comparing FPGA implementations developed and reported by different people, and that target different architectures, often yields only a very rough idea of relative performance, along with a long list of caveats. In addition, without context the outcome of this comparison is not very useful. There are several causes for this. *Bundled resources*, such as "slices" and "logic elements", are often inappropriately used as a metric which leaves room for interpretation. The "definition" of these bundlings also change with time and are inconsistent across device families, even ones from the same vendor. The Virtex-5 "slice", for example, has twice as many LUTs and FFs as a Virtex-4 "slice" in addition to having a 6-input LUT instead of 4-input one. Even without these differences, when "slices" are used as a fundamental unit, we cannot know the exact amount of resources that are actually being used — a single flip-flop, two, four, or one and two LUTs? In addition, "slices" do not include additional resources that are sometimes used for cipher implementations such as BRAMs and DSPs. This means that comparing designs on "slice-count" alone does not make sense except in the case where a design is packed into fewer slices of the same device, using the same software tools, while maintaining or exceeding throughput. Since all resources within a "slice" share signals such as clock and enable signals, they cannot be used by functions in a different clock domain; thus, better packing frees up otherwise unusable resources, and is indicative of the logic-packing efficiency and skills of the designer (and/or tools). Using LUTs as a fundamental unit is problematic in a similar way since not all of its memory cells are used. Our unrolled module, for example, uses 672 6-input LUTs, which would translate into $672 \cdot 64 = 43,008$ memory cells, where in fact, all these LUTs are used as either 2 or 3 bit shift registers consuming only $1,696$ memory cells (this is shown in Table 2 as "distributed RAM", d.RAM). XST reports the distribution of LUT RAM bits and with these figures we are able to more accurately compare designs across architectures. Due to the above reasoning, it is now also apparent that the often-used "throughput per slice" metric of performance is unsuitable for comparison purposes, and should be used only in special cases; our 55 Gbit/s fully unrolled design uses only 428 "slices" — about 128 Mbit/slice — though, clearly, we cannot fairly use this figure in comparison with other designs. One way to hold some variables constant is to implement designs from multiple sources from code, while using the same tools and targeting the same architecture. Results from this approach, however, may be skewed by an *asymmetry of effort* because implementers know their own designs best, and may naturally end up investing more effort in the optimizing them. When it comes to cipher implementations, such

| Design | Dec[a] | Key sch. | Device | Resources[b] | | | | | | $f$ | Throughput[c] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | slices | LUT | FF | d.RAM | BRAM | DSPs | (MHz) | (Gbit/s) |
| *Basic* | ○ | ○ | Virtex-5 | 93 | 245 | 274 | 7838 | 2×36K | 4 | 550 | 1.76 |
| *Round* | ○ | ○ | Virtex-5 | 277 | 204 | 601 | 1432 | 8×36K | 16 | 485 | 6.21 |
| *Unrolled* | ● | ○ | Virtex-5 | 428 | 672 | 992 | 1696 | 80×36K | 160 | 430 | 55 |
| Algotronix [1] | ○ | ○ | Virtex-5 | 161 | *n/a* | *n/a* | *n/a* | 2×36K | 0 | 250 | 0.8 |
| Chaves *et al.* [2] | ● | ○ | Virtex-II Pro | 515 | *n/a* | *n/a* | *n/a* | 12×18K | 0 | 182 | 2.33 |
| Helion [7] | ○ | ● | Virtex-5 | 349 | *n/a* | *n/a* | *n/a* | 0 | 0 | 350 | 4.07 |
| Kotturi *et al.* [10] | ● | ○ | Virtex-II Pro | 10816 | *n/a* | *n/a* | *n/a* | 400×18K | 0 | 126 | 16 |
| Hodjat *et al.* [8] | ○ | ○ | Virtex-II Pro | 5177 | *n/a* | *n/a* | *n/a* | 84×18K | 0 | 168 | 21.5 |
| Chaves *et al.* [2] | ● | ○ | Virtex-II Pro | 3513 | *n/a* | *n/a* | *n/a* | 80×18K | 0 | 272 | 34.7 |

[a]For "basic" and "round" implementations, decryption can be achieved by adding 32 bit muxes in the datapath between BRAM and DSP.
[b]Virtex-5 has 4 FF and 4 6-LUT per slice and a 36 Kbit BRAM, while Virtex-II PRO has 2 FF and 2 4-LUT per slice, and an 18 Kbit BRAM.
[c]For "basic" and "round" implementations, figures reflect two and eight concurrent stream processing, respectively.

**Table 2. Our results along with recent academic and commercial implementations.**

as AES, the *variety of modes* makes it such that designers rarely implement those with identical set of functions (encryption/decryption or both, inclusion of key schedule, key-lengths supported, support of various modes of operation, and so on). Standaert [13] discusses this as well, and poses nine questions to ask when implementing AES with respect to design goals. *Implementation conditions* such as which software tools, speed grades, verification/simulation conditions, and the stage from which figures were quoted (post-synthesis, map, or place-and-route) are often missing, but are crucially important for evaluation. Post-synthesis performance report, for example, can be significantly different (often too optimistic) than the post place-and-route report that take actual delays into account (synthesis reported that our unrolled variant will run at 655 MHz). Finally, the *unavailability of source code* of many designs is detrimental to the process of effective evaluation and comparison.

So how can we better compare designs? We should start by mandating the reporting of all possible details, as we have tried to do in this paper, and encourage making the source code available for evaluation. When releasing code is not possible, synthesis, map, PAR, and timing reports should be accessible instead. The best way to compare, however, is to assess the suitability of one design over another in meeting the constraints of a specific end application. An end application provides context, without which, the merits of each design cannot be fully appreciated. System designers would benefit the most from our implementations if their application requires processing multiple streams, that other FPGA functions do not use all BRAMs and DSP blocks, and that other logic resources are scarce. We report the performance results for our designs when they are used as stand-alone functions and under optimal conditions, though when used in conjunction with other functions these maximum throughput figures may degrade. For example, if other functions occupy many of the resources (routing, slices), the efficient routing we use as a stand-alone function may no longer be fully available. Other functions that interface with these AES modules may not be capable of matching their operating frequency, so in these cases, either more elements need to be added, such as FIFOs, or other proposed AES designs may be better suited. We now survey previous work.

## 6.1 Prior work

McLoone *et al.* [12] proposed an implementation of the AES-128 in ECB mode based on the Xilinx Virtex-E 812(-8) device using 2,457 "Configurable Logic Blocks" and 226 BRAMs providing an overall encryption rate of 12 Gbit/s. Hodjat *et al.* [8] report an AES-128 implementation with 21.54 Gbit/s throughput using 5,177 "slices" and 84 BRAMs on a Xilinx Virtex-II Pro 20(-7) FPGA. Zhou *et al.* [19] recently reported figures for an AES implementation of the Galois Counter Mode (GCM) on a Xilinx Virtex-4 LX40(-12) FPGA achieving a throughput of 20.6 Gbit/s without use of BRAMs, which uses 8,035 "slices". Gaj and Chodowiec [6] proposed an area-efficient AES implementation on a Xilinx Spartan-II 30(-6) with 222 "slices" and 3 BlockRAMs with an encryption rate of 0.166 Gbit/s. Fischer and Drutarovský [5] proposed an economic and a high-performance AES implementation on an Altera ACEX 1K100(-1) device FPGAs using the T-Table technique. Their economic encryptor/decryptor yielded a throughput of 0.212 Gbit/s using 12 "Embedded Array Block" memories and 2,923 "Logical Elements". The fast implementation based on an Altera APEX 1K400(-1) and T-tables requires 86 "Embedded System Block" memories and 845 "Logical Elements" to provide a throughput of 0.750 Gbit/s. Chaves *et al.* [2] also use the memory-based AES implementation with a Virtex-II Pro 20(-7) where an architecture implementing a single iteration and a loop unrolled AES based on a similar strategy as ours.

Following the discussion in this section, we are able to conclude that comparing designs based on a couple of figures of merit without the context of an application does not do justice to any of the compared designs. We thus provide Table 2 only as a reference, with designs that are closest to the ones we report in this paper. Since we are unaware of other Virtex-5 AES implementations in the academic literature, we also consider two commercial AES cores by Algotronix [1] and Helion Technology [7].

## 7 Conclusions and future work

We have presented new ways for performing AES operations at a high throughput using on-chip RAM and DSP blocks with minimal use of traditional user logic such as flip-flops and look-up tables. The source code for these designs are publicly available so that they are used in further research, and provide readers with the ability to reproduce our reported results.

Future work includes integrating the key scheduling operation into the architecture while maintaining throughput and exploring efficient ways in which to use the basic module for further modes of operation. We would also like to give these designs context and integrate them into a real application, and measure performance and power consumption on an actual device. Another interesting direction is the evaluation of our designs against side channel attacks, such as power analysis. Since we use the same basic construct for our designs, we would be able to evaluate how pipelining and unrolling affect power signatures, similarly to what was previously done by Standaert *et al.* [14], but on newer architectures.

## Acknowledgments

We thank Xilinx for the donation of development tools, and David Ellington for helping resolve simulation issues. Saar Drimer's research is funded by Xilinx Inc.

## References

[1] Algotronix Ltd. AES G3 data sheet Xilinx edition, October 2007. `http://www.algotronix-store.com/kb_results.asp?ID=7`.

[2] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa. Reconfigurable memory based AES co-processor. *Reconfigurable Architectures Workshop*, page 192, 2006.

[3] M. Dworkin. *SP 800-38C: Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality.* U.S. NIST, 2005.

[4] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 9(4):545–557, 2001.

[5] V. Fischer and M. Drutarovský. Two methods of Rijndael implementation in reconfigurable hardware. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162, pages 77–92, 2001.

[6] K. Gaj and P. Chodowiec. Very compact FPGA implementation of the AES algorithm. In *CHES*, volume 2779, pages 319–333, 2003.

[7] Helion Technology Ltd. High performance AES (Rijndael) cores for Xilinx FPGAs, 2007 (Rev. 2.3.3). `http://www.heliontech.com/downloads/aes_xilinx_helioncore.pdf`.

[8] A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In *Field-Programmable Custom Computing Machines*, pages 308–309. IEEE Computer Society, 2004.

[9] T. Ichikawa, T. Kasuya, and M. Matsui. Hardware evaluation of the AES finalists. *AES Candidate Conference*, pages 13–14, 2000.

[10] D. Kotturi, S.-M. Yoo, and J. Blizzard. AES crypto chip utilizing high-speed parallel pipelined architecture. In *IEEE International Symposium on Circuits and Systems*, pages 4653–4656, 2005.

[11] M. McLoone and J. McCanny. High performance single-chip FPGA Rijndael algorithm implementations. In *CHES*, volume 2162, pages 65–76, 2001.

[12] M. McLoone and J. McCanny. Rijndael FPGA implementations utilising look-up tables. *The Journal of VLSI Signal Processing*, 34(3):261–275, 2003.

[13] F.-X. Standaert. Secure and efficient implementation of symmetric encryption schemes using FPGAs, 2007. `http://www.dice.ucl.ac.be/~fstandae/PUBLIS/45.pdf`.

[14] F.-X. Standaert, S. B. Örs, and B. Preneel. Power analysis of an FPGA implementation of Rijndael: Is pipelining a DPA countermeasure? In *CHES*, volume 3156 of *LNCS*, pages 30–44, London, UK, 2004. Springer.

[15] F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat. Efficient implementation of Rijndael encryption in reconfigurable hardware: improvements and design tradeoffs. In *CHES*, volume 2779, pages 334–350, 2003.

[16] U.S. National Institute of Standards and Technology (NIST). *FIPS 197: Advanced encryption standard*, 2001.

[17] Xilinx Inc. *UG190: Virtex-5 user guide*, 2006.

[18] Xilinx Inc. *UG193: Virtex-5 XtremeDSP design considerations user guide*, 2007.

[19] G. Zhou, H. Michalik, and L. Hinsenkamp. Efficient and high-throughput implementations of AES-GCM on FPGAs. In *Field Programmable Technology*, pages 185–192, 2007.