Efficient Utilization of DSPs and BRAMs Revisited: New AES-GCM Recipes on FPGAs

(Full Version)

Elif Bilge Kavun
The University of Sheffield, Sheffield, UK
e.kavun@sheffield.ac.uk

Nele Mentens
imec-COSIC and ES&S, ESAT, KU Leuven, Leuven, Belgium
nele.mentens@kuleuven.be

Jo Vliegen
imec-COSIC and ES&S, ESAT, KU Leuven, Leuven, Belgium
jo.vliegen@kuleuven.be

Tolga Yalçın
Northern Arizona University, Flagstaff, AZ, US
tolga.yalcin@nau.edu

***Abstract*—In 2008, Drimer et al. proposed different AES implementations on a Xilinx Virtex-5 FPGA, making efficient use of the DSP slices and BRAM tiles available on the device. Inspired by their work, in this paper, we evaluate the feasibility of extending AES with the popular GCM mode of operation, still concentrating on the optimal use of DSP slices and BRAM tiles. We make use of a Xilinx Zynq UltraScale+ MPSoC FPGA with improved DSP features.**

For the AES part, we implement Drimer's round-based and unrolled pipelined architectures differently, still using DSPs and BRAMs efficiently based on the AES Tbox approach. On top of AES, we append the GCM mode of operation, where we use DSP slices to support the GCM finite field multiplication. This allows us to implement AES-GCM with a small amount of FFs and LUTs. We propose two implementations: a relatively compact round-based design and a faster unrolled design.

I. Introduction

Field Programmable Gate Arrays (FPGAs) are not what they were when first introduced in the mid 1980s. The main components of the first FPGAs were gates that could be reconfigured to different logical functions. Today, almost 40 years later, industrial vendors make very heterogeneous devices that contain dedicated cores on the silicon die, next to the traditional reconfigurable gates. These dedicated cores, hard cores, include memory cores, cores for Digital Signal Processing (DSP), Analog-to-Digital Converters, and communication cores. Earlier this decade, both major vendors, Xilinx and Intel, launched FPGAs that combine the reconfigurable hardware with industry-standard processors. Knocking on the door, are cores that facilitate Artificial Intelligence optimized calculations.

Although such a versatile array of different cores makes that for every application there is a fit, it also means that some dedicated cores are unused, while others are intensively used. In order to optimize the occupation of the dedicated cores, it is a challenge for the hardware designer to use the dedicated cores for applications that they were not originally intended for.

This paper is inspired by the work of Drimer et al. [1], in which the authors implemented the Advanced Encryption Standard (AES) [2] on a Xilinx Virtex-5 device, mostly using DSP slices and Block RAM (BRAM) cores. Since symmetric ciphers are almost always used in combination with a mode of operation, we extend the work of Drimer et al. by implementing both AES and the Galois/Counter Mode (GCM) [3], still concentrating on maximizing the use of DSP slices and BRAM cores. Our implementation is targeted to a recent 16 nm Xilinx Virtex UltraScale family member.

The main contributions of this work are:

1) The implementation of round-based and unrolled pipelined AES architectures using BRAMs and DSP slices in an optimal way while minimizing the use of flip-flops (FFs) and look-up tables (LUTs) (improving Drimer et al.'s work [1]),
2) The addition of the GCM mode of operation to the AES core by implementing the binary multiplier on the DSP slices of the Xilinx Virtex UltraScale FPGA [4] efficiently, supporting both round-based and unrolled AES architectures.

The remainder of the paper is organized as follows: Section II summarizes related work. The implemented algorithms are discussed in Section III. Section IV describes the implementations in more detail and Section V discusses our results. Finally, Section VI concludes the paper.

II. Related Work

Drimer et al., in [1], present a new recipe for implementing AES on an FPGA. Their work is tailored for applications in which the regular reconfigurable gates are scarce, yet not all BRAM and DSP blocks are used. They focus on a Xilinx Virtex-5 FPGA of which they use the dual-port BRAMs and the DSP slices. Their implementation uses the T-table approach, which is an optimization technique for AES, especially useful on 32-bit platforms, introduced by the designers of AES [5]. Drimer et al. are able to implement AES on a Xilinx Virtex-5 using 8 36K dual-port BRAMs and 16 DSPs for the round-based AES implementation and 80 36K dual-port BRAMs and 160 DSPs for the unrolled pipelined version (a number of LUTs and FFs used as well).

Other work on the implementation of cryptographic algorithms on FPGAs using DSP slices and BRAMs, includes the design of architectures for Elliptic Curve Cryptography [6], lattice-based schemes [7] and hash functions [8].

III. Algorithms

A. AES

The Rijndael block cipher was selected by the National Institute for Standard and Technology (NIST) to be the new encryption algorithm standard after an open competition that ended in 2000 [2].

AES has a block length of 128 bits and a variable key length. For encryption, the plaintext is processed in 10, 12 or 14 rounds for key lengths of 128, 192 or 256 bits, respectively. Each round consists of four operations:

- *SubBytes* – where 16 bytes of the 128-bit input are updated according to a 8-bit look-up table,
- *ShiftRows* – where the rows of the AES state are cyclically shifted by certain offsets,
- *MixColumns* – where the four bytes of each column of the AES state are combined using an invertible linear transformation, and
- *AddRoundKey* – where the round key is XOR-ed to the AES state.

In the last round, the MixColumns operation is omitted, and before the first round, the encryption starts with an additional AddRoundKey operation.

The AES SubBytes step can be realized as a finite field multiplication or as look-up table(s). In the look-up table setting, it is also possible to append the MixColumns step (even together with ShiftRows) to the look-up table and redefine the table as 8-bit input, 32-bit output instead of the original 8-bit input, 8-bit output table. The Sbox output is normally defined as $y \to S(x)$, but the T-table is defined as $y \to (2S(x), S(x), S(x), 3S(x))$ (or as its shifted versions due to ShiftRows) as a result of the multiplication with the MixColumns coefficients.

In our AES core implementations, we are following the T-table approach.

B. GCM

GCM is a recommended mode of operation for symmetric-key block ciphers [3], capable of achieving authenticated encryption with associated data. This is done by using the block cipher in counter mode (CTR) and the GHASH function, as shown in Fig. 1.

The ciphertext is generated as the bitwise XOR of the plaintext and the output of a block cipher, which has a counter value at its input, initialized by an initialization vector (IV). The authentication tag in GCM is obtained from the keyed hash function GHASH, as standardized in [3] and shown in Fig. 2.

The modular multiplier in GHASH performs the operation $(X \oplus I) * H$, where X feeds back the result of the modular multiplication.

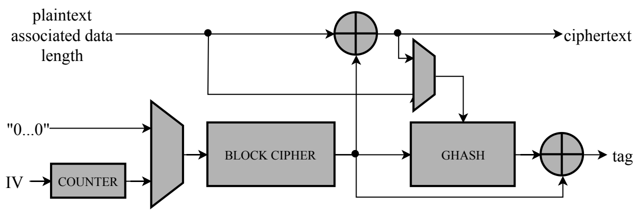


Fig. 1. Top-level architecture of the Galois/Counter Mode of operation, where $oplus$ denotes an XOR and a rectangle with a cross inside denotes a register

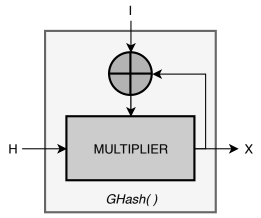


Fig. 2. Functional representation of the operation in GHASH

The input I is generated by segmenting the concatenation of

1) the (possibly padded) associated data,
2) the (possibly padded) ciphertext, and
3) the lengths of both the associated data and the ciphertext.

The input H is equal to the ciphertext that comes out of the block cipher when the plaintext consists of all zeros. This means that H stays constant as long as the key does not change.

It is pointed out that the multiplication is done in the binary extension field GF(2^{128}) with $x^{128} + x^7 + x^2 + x + 1$ as the irreducible polynomial.

Finally, the tag is generated by XOR-ing the output of the GHASH function with the ciphertext that comes out of the block cipher when the plaintext is equal to the IV.

This work elaborates on

1) the BLOCK CIPHER (AES) in Fig. 1, and
2) the MULTIPLIER in Fig. 2.

Both components are integrated into a 128-bit AES-GCM implementation.

IV. Implementation

The FPGA implementation consists of a 128-bit AES core and a GF(2^{128}) multiplier in combination with control logic, multiplexers and registers. The overall AES-GCM architecture is depicted in Fig. 3. In the following paragraphs, we explain the implementation details of the AES core and the GF(2^{128}) multiplier.

The BLOCK CIPHER in the architecture image contains the AES implementations, described in Sect. IV-A while the MULTIPLIER is discussed in Sect. IV-B. Furthermore, there are three 128-bit registers (represented by rectangles with crosses inside), a COUNTER, two multiplexers, and control logic in the form of a Finite State Machine (FSM). These

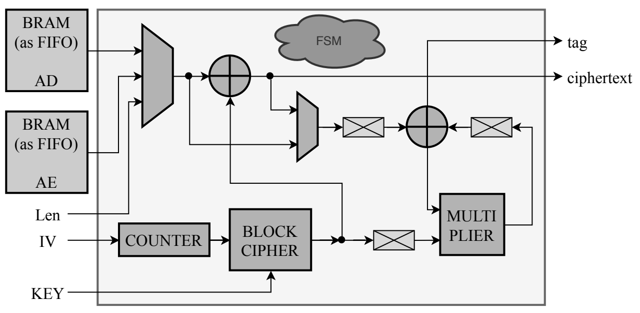


Fig. 3. The overall AES-GCM architecture

components, that form the core of the implementation, are evaluated in this paper and are surrounded by a rectangle in Fig. 3. The Associated Data (AD) and the data for Authenticated Encryption (AE) are applied to the AES-GCM core through FIFOs implemented in BRAM. The other inputs to the core are applied directly.

A. AES

Our AES core implementation follows a very similar approach to Drimer et al.'s work: using T-tables in order to come up with a RAM-based implementation. This way, it is possible to use the existing BRAM resources on an FPGA efficiently while minimizing LUT and FF utilization. Furthermore, as in the case of Drimer et al., DSP slices are also used to realize certain AES steps which again results in less LUT utilization. Even though our work is inspired by Drimer et al., our implementation differs from theirs in many ways.

1) Differences to previous work: In Drimer et al.'s work, the AES core takes 32-bit data as input and the key scheduling is not covered in the design. They use 36K (18K+18K) dual-port BRAMs to store T-tables (8 36K BRAMs used in a round-based architecture and 80 36K BRAMs used in an unrolled pipelined architecture). For regular rounds, they define 4 different types of T-tables ($T0$, $T1$, $T2$, $T3$ - each 8K) using different MC multiplication coefficients (shifted versions of 2, 3, 1, 1). Out of these, they place only $T0$ and $T2$ in the BRAMs and shift their outputs later in DSP slices to get the $T1$ and $T3$ values. For the last round, they define $T0'$, $T1'$, $T2'$, and $T3'$ T-tables which are basically the regular Sboxes for skipping the MixColumns step, and these are placed in the second half of each BRAM (18K out of 36K). As a result, the organization of each of the 36K dual-port BRAMs looks like:

$T0$
$T0'$

and

$T2$
$T2'$

In our case, the AES core takes 128-bit data and key as inputs. We define two types of T-tables for data substitution, T and T' (8K each), where T and T' correspond to the concatenation of $2S$, S, S, $3S$ and S, 0, 0, 0, respectively. Note that S is the original Sbox content, whereas kS is the contents of Sbox multiplied by k over the finite field. The organization of each 18K dual-port BRAMs then looks like:

$2S$	S	S	$3S$
S	0	0	0

We furthermore utilize BRAM and DSP blocks in order to realize key scheduling in our design. Unlike Drimer et al, we do not need decryption (thanks to GCM mode), which enables us to implement the last round without using any additional logic.

2) Round-based AES core: The block diagram of the AES core can be seen in Fig. 4.

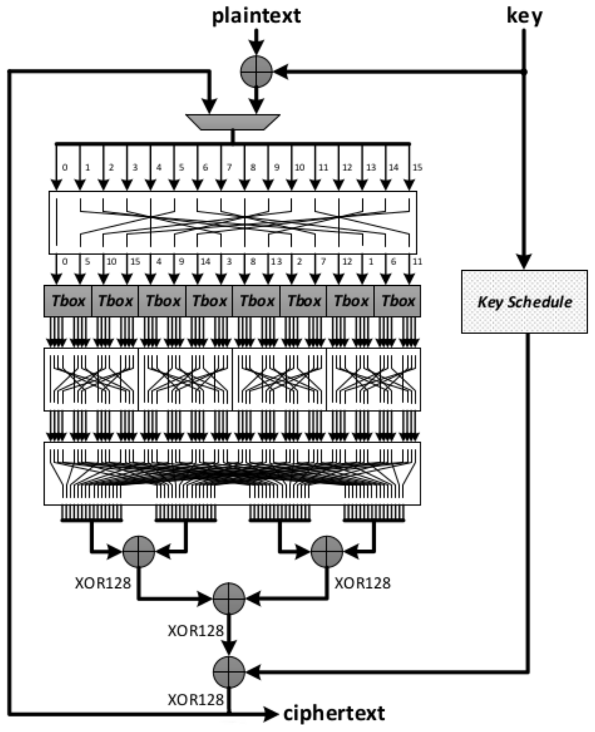


Fig. 4. Block diagram of the AES core

In our implementation, we exchange the order of SubBytes and ShiftRows operations, which has no effect on the overall functionality. We start by splitting the 128-bit state into 8-bit (1-byte) chucks and then applying ShiftRows on these bytes, which in fact is only re-ordering of bytes. In physical terms, this corresponds to renaming the nets. Following this, we supply each byte to its corresponding Tbox, in order to obtain the corresponding Sbox outputs, S, and their multiplied versions kS. Each adjacent 4 Tbox outputs correspond to a column, and parts of each Tbox output within the column are combined (added) to form the corresponding MixColumns output. In the last round, the second half of the Tboxes are read from RAM, which when combined with the bytes from neighboring Tboxes generate just the regular Sbox output. In other words, MixColumns operation is skipped in the last round in conjunction with the AES specification.

Let's see how MixColumns is realized by looking at the first column. The inputs to the first column Tboxes are bytes 0, 5, 10 and 15 of the state. The outputs of the 4 Tboxes are then t_0, t_1, t_2 and t_3, each being 4-byte wide with the following expressions:

$$
\begin{aligned}
t_0 &= \{t_{00}, t_{01}, t_{02}, t_{03}\} = \{2s_0\ , s_0\ , s_0\ , 3s_0\ \} \\
t_1 &= \{t_{10}, t_{11}, t_{12}, t_{13}\} = \{2s_5\ , s_5\ , s_5\ , 3s_5\ \} \\
t_2 &= \{t_{20}, t_{21}, t_{22}, t_{23}\} = \{2s_{10}, s_{10}, s_{10}, 3s_{10}\} \\
t_3 &= \{t_{30}, t_{31}, t_{32}, t_{33}\} = \{2s_{15}, s_{15}, s_{15}, 3s_{15}\}
\end{aligned}
$$

The first 4 MixColumns outputs are then generated using

$$
\begin{aligned}
m_0 &= t_{00} + t_{11} + t_{22} + t_{33} = 2s_0\ + s_5\ + s_{10} + 3s_{15} \\
m_1 &= t_{10} + t_{21} + t_{32} + t_{03} = 2s_5\ + s_{10} + s_{15} + 3s_0 \\
m_2 &= t_{20} + t_{31} + t_{02} + t_{13} = 2s_{10} + s_{15} + s_0\ + 3s_5 \\
m_3 &= t_{30} + t_{01} + t_{12} + t_{23} = 2s_{15} + s_0\ + s_5\ + 3s_{10}
\end{aligned}
$$

In practice, prior to addition, the first bytes of all Tbox outputs are combined to form the 128-bit "first-byte" vector, the second bytes to form the 128-bit "second-byte" vector, and so on. These vectors are then added using 128-bit XOR blocks to generate all MixColumns outputs at once. Each 128-bit XOR block is formed of two 40-bit XOR blocks and 48-bit XOR block, each of which is implemented using the DSP48 modules in "combinational" function mode.

For the key scheduling part, additional BRAMs are required due to 4 Sbox calls in the last 32-bit word of the round key. The rc values for the 8-bit round constant addition is also stored in BRAM. We furthermore use BRAMs instead of registers by storing a one-to-one mapping lookup table, which we call "Bbox", i.e. $B(x) = x$. The block diagram describing AES key scheduling can be seen in Fig. 5.

The output of each round is sent to the input of next round together with the generated round key.

The round operation is repeated for 10 rounds; at the end of 10 rounds, "done" flag is generated and the ciphertext is read at the round output.

In our implementation, only LUTs and FFs are used by the control logic.

It should also be noted that dual-port nature of BRAMs allows us to utilize each BRAM as a dual T/S/Bbox, where the two address ports are the box inputs and their corresponding output ports are the box outputs.

3) Unrolled pipelined AES core: The flow of the pipelined unrolled AES core is the same as the round-based, only difference is that all rounds are implemented in an unrolled

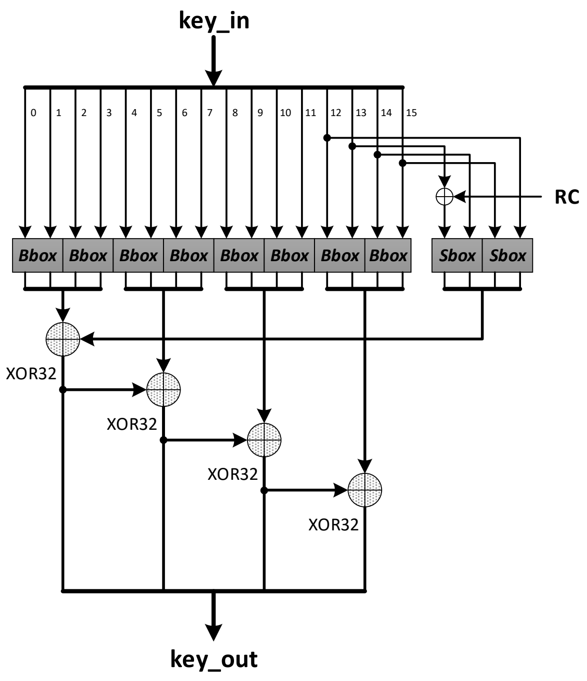


Fig. 5. AES key scheduling

pipelined fashion for faster execution with more area utilization. Since all round constants are fixed in the unrolled version, there is no need for the control logic. As a result, this is a "zero-LUT/FF" design, where only BRAM and DSP blocks are utilized.

4) Resource utilization: In the round-based design, mostly BRAMs, DSPs, and some number of LUTs and FFs are utilized. 8 36K dual-port BRAMs are required for the data processing part (as in Drimer et al.' work) and 10 18K BRAMs (5 36K BRAM tiles) are required for the key scheduling. In addition, as round constants are also defined in BRAM, we need an additional 18K BRAM (0.5 36K BRAM tile). In detail, the contents of the BRAMs are as follows: Data BRAM – 512 addresses x 32-bit width = 16K (8K+8K) bits, Key BRAM – 512 addresses x 8-bit width = 4K (2K+2K) bits, RC BRAM – 10 addresses x 8-bit width = 80 bits.

3 DSPs are used for 1 128-bit XOR operation and there are 4 128-bit XOR operations per round, which results in 15 DSPs together with the initial 128-bit key addition. For 4 32-bit XOR operations in key scheduling, 5 DSPs are used. Additionally, 4 FFs for the round counter and LUTs for multiplexing logic are used.

The resource utilization for unrolled pipelined version is exactly 10 times of round-based cost, 135 BRAMs and 173 DSPs (3+12x10+5x10). No FFs or LUTs are necessary as there is no control logic for unrolled design.

B. GF(2^{128}) Multiplier

The implementation of the GF(2^{128}) multiplier follows Algorithm 1, which processes one of the operands in a serial way, starting with the most significant bit (MSB), similar to the architecture proposed by Sakiyama et al. in [9]. Our aim is to implement Algorithm 1 making optimal use of the DSP slices in the targeted FPGA. This comes down to mapping the computation in the for-loop in the algorithm (except for the multiplication with x, which is a shift operation) onto the DSP slices.

```
Require: $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$,
         $A(x) = \sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{m-1} b_i x^i$
Ensure : $T(x) = A(x) \cdot B(x) \mod P(x)$
$T(x) = \sum_{i=0}^{m} t_i x^i \leftarrow 0$
for $i = m - 1$ to 0 do
    $T(x) \leftarrow (T(x) + a_i \cdot B(x) + t_m \cdot P(x)) \cdot x$
end
$T(x) \leftarrow T(x)/x$
```

Algorithm 1: MSB-first bit-serial modular multiplication over GF(2^m)

Fig. 6 shows a simplified version of the DSP48E2 slice in a Xilinx UltraScale+ FPGA. Note that many multiplexers and registers have been omitted from the figure for clarity reasons. The DSP slice consists of two stages. First, there is a dedicated multiplier (denoted with X) of which one of the operands comes from a pre-adder/subtracter (denoted with +/-). In the second stage, there is an addition/subtraction/logic block (denoted with +/-/l), which can perform different arithmetic and logic operations. Whereas the first stage uses the inputs of the DSP slices ($\mathtt{DSP}_1$, $\mathtt{DSP}_2$, $\mathtt{DSP}_3$ and $\mathtt{DSP}_4$, indicated with black triangles), the second stage gets its inputs from three multiplexers ($\mathtt{MUX}_X$, $\mathtt{MUX}_Y$ and $\mathtt{MUX}_Z$). Note that each multiplexer can also be configured to pass through a value that consists of all zeros. The output of the +/-/l block is stored in a register and connected to the output of the DSP slice, $\mathtt{DSP}_P$ (indicated with a black triangle). Direct connections are made with neighboring DSP slices on the top and bottom of the slice (represented by white triangles). These connections cannot be reached through regular routing.

The DSP slices on an UltraScale+ FPGA can have the +/-/l operator in the second stage perform a three-input XOR. This new feature, which is not available in the 7-series FPGAs, is utilized to perform the 3-input addition in the for-loop in Algorithm 1. The other operations in the for-loop are a multiplication with x, i.e. a shift operation that is handled through the routing outside of the DSP slice, and multiplications of a GF(2) element with a GF(2^{128}) element, namely $a_i \cdot B(x)$ and $t_m \cdot P(x)$. We explore two different options to implement these multiplications. The first option is to use a multiplexer that has the GF(2) element at its selection input and the GF(2^{128}) element at one of its data inputs, while the second data input is fed with zeros. This leads to the representation in Fig. 7, showing the two multiplications

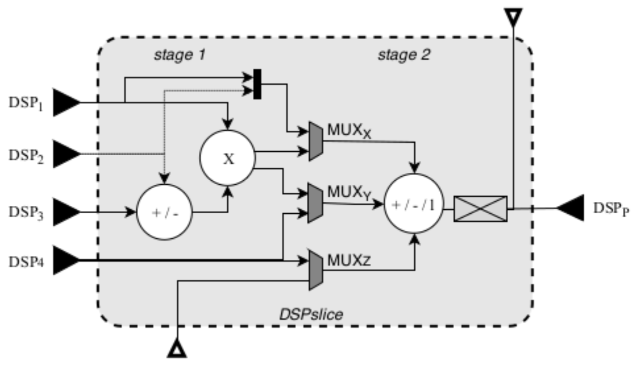


Fig. 6. Functional representation of a Xilinx DSP48E2 slice

through two multiplexers and the addition through a three-input XOR. The second option is to use the multiplier in stage 1 of the DSP slice. Although this is an integer multiplier and we need to perform a multiplication in GF(2^{128}), the product is correct, since one of the operands consists of only one bit. This bit is padded with zeros. We explore the feasibility of both multiplier options in the following paragraphs.

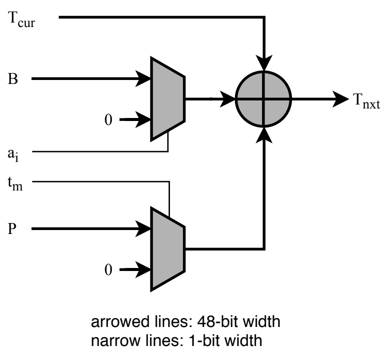
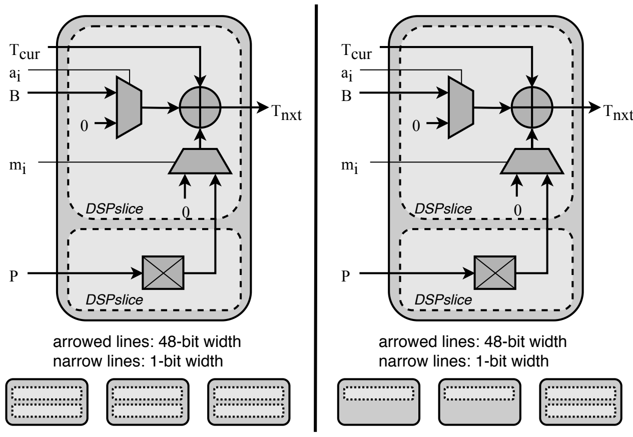


Fig. 7. A functional representation of the multiply and add approach using DSP slices

1) GF(2^{128}) multiplication using an XOR and multiplexers: In order to map the 128-bit operations in the for-loop of Algorithm 1 on the 48-bit DSP slices, we need at least three slices ($3 * 48 > 128$). For the first option described above, shown in Fig. 7, each three-input XOR is mapped on the +/-/l operator in a DSP slice, as depicted in Fig. 8 on the left side. The left input of the XOR comes from $\mathtt{MUX}_X$ and the top input is mapped to $\mathtt{MUX}_Y$. That means that the value B in Algorithm 1 is connected to the concatenation of the inputs $\mathtt{DSP}_1$ and $\mathtt{DSP}_2$, and the value T is connected to input $\mathtt{DSP}_4$. The bit a_i is used as the selection signal for $\mathtt{MUX}_X$, choosing between B and zero. Because input $\mathtt{DSP}_4$ is already routed to $\mathtt{MUX}_Y$, it cannot be used for $\mathtt{MUX}_Z$. Therefore, the only option that is left to connect P to $\mathtt{MUX}_Z$, is to route P through the neighboring DSP slice. As a result, the number of DSP slices is doubled, leading to three pairs of DSP slices, i.e. six DSP slices in total, for the computation of the for-loop in

Algorithm 1. This is shown below the architectural DSP slice mapping on the left side of Fig. 8.



Tcur
ai
B
0
Tnxt
mi
DSPslice
0
P
DSPslice
arrowed lines: 48-bit width
narrow lines: 1-bit width
Tcur
ai
B
0
Tnxt
mi
DSPslice
0
P
DSPslice
arrowed lines: 48-bit width
narrow lines: 1-bit width

Fig. 8. The construction of Cells

Because P is a fixed value (P is the irreducible polynomial that is fixed in the GCM specification), we use the output registers of the three additional DSP slices to store P, as indicated with a rectangle with a cross inside in Fig. 8 on the left side. This way, the connection of P does not introduce a routing delay.

On the right side of Fig. 8, an optimized version is shown, that takes into account that only the least significant bits of P contain values different from zero, which means the two DSP slices dealing with the 96 most significant bits of P are not needed. As opposed to the architecture on the left side of Fig. 8, which works for any irreducible polynomial P, the architecture on the right side only works with $P = x^{128} + x^7 + x^2 + x + 1$, as specified in the GCM standard.

2) GF(2^{128}) multiplication using an XOR and multipliers: The maximum input width of the multiplier block in the DSP slice is $27 * 18$, which means we need five DSP slices to perform the multiplication $B \cdot a_i$ with input width $128 * 1$. When the multiplier block is used in the first stage of the DSP slice, the +/-/l operator can only be configured as an adder or a subtracter, which means the multiplier cannot be combined with a three-input XOR in the same DSP slice. Therefore, we need five DSP slices for the multiplier and five more DSP slices for the 128-bit three-input XOR. Assuming that the multiplication $P \cdot t_m$ is implemented using a multiplexer, we would need 10 DSP slices to execute the for-loop in Algorithm 1 using this construction. That is a lot more than the four DSP slices required for the GF(2^{128}) multiplication using an XOR and multiplexers. Therefore, we only use the option shown in Fig. 8 on the right side.

In order to extend the implementation in Fig. 8, the for-loop needs to be executed 128 times. In an AES-GCM implementation where the AES core generates a new 128-bit ciphertext block every cycle (which is the case for the unrolled pipelined AES core), the GF(2^{128}) multiplication needs to be done in one clock cycle. In an AES-GCM implementation where the AES core is round-based, i.e. a new 128-bit ciphertext block is generated every 10 cycles, the GF(2^{128}) multiplication needs to be completed within 10 clock cycles. We implement a 1-cycle as well as a 10-cycle architecture of the GF(2^{128}) multiplier. In the 1-cycle architecture, the for-loop in Algorithm 1 is fully unrolled without pipelining. In the 10-cycle architecture, the for-loop in Algorithm 1 is partially unrolled, i.e. 13 instances of the loop computation are instantiated, followed by a register and a feedback signal. This way, after 10 iterations, all 128 a_i bits are processed.

In summary, to construct a stack of cells that can perform a single 128-bit wide modular multiplication in a single clock cycle, 512 (128×4) DSP slices are required. Analogously, when performing one operation in 10 clock cycles, 52 (13×4) DSP slices are required.

V. Results

We evaluate the implementation properties of our AES-GCM architectures using Xilinx Vivado 2017.3 suite after placement and routing. The target platform is ZCU102 evaluation kit which contains a Zynq UltraScale+ FPGA. As explained in Section IV, we implement two different architectures, namely a round-based architecture and a fully unrolled architecture. The results for both AES-GCM implementations are shown in Table I, as well as the individual results for the AES cipher and the GF(2^{128}) multiplier. The glue logic represents the additional registers, multiplexers and FSM.

TABLE I
Implementation results of AES-GCM on ZCU102

	LUT	FF	BRAM	DSP	T_{min}(ns)
unrolled PL	899	1036	139	685	172
AES	192	0	135	173	
MULTIPLIER	325	401	0	512	
glue	682	635	4	0	
round based	785	1043	17.5	72	20
AES	196	4	13.5	20	
MULTIPLIER	156	398	0	52	
glue	433	641	4	0	

With respect to LUT and FF utilization, our design drastically outperforms the smallest AES-GCM implementation on FPGA, presented by Zhou et al. in [10]. The architecture in [10] reports 4628 slices on a Virtex-5 FPGA. Knowing that each slice contains 4 LUTs and 4 FFs, our implementation reduces the occupation of the LUTs and FFs by a factor 20 thanks to the optimal utilization of DSP slices and RAM blocks.

The timing of the round-based architecture is shown in Fig. 9. After the calculation of H, the associated data are processed in chunks of 128 bits. Then, the AES core encrypts the plaintext blocks, and the multiplier uses the ciphertext to generate the tag. The last step in the generation of the tag, is the use of the length field and the addition of the initial encryption of the IV. In total, this leads to a latency of:

- 12 clock cycles to calculate H (10 cycles for AES plus 2 cycles overhead)

- 12 clock cycles per 128-bit block of associated data
- 12 clock cycles per 128-bit block of plaintext data
- 12 clock cycles to synchronize the pipeline
- 12 clock cycles to process the length field
- 1 clock cycle to finalize the generation of the tag

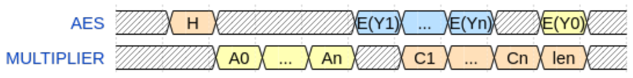


Fig. 9. Round-based timing

For the unrolled pipelined architecture, we need:

- 1 clock cycle to calculate H
- 1 clock cycle per 128-bit block of associated data
- 1 clock cycle per 128-bit block of plaintext data
- 1 clock cycle to synchronize the pipeline
- 1 clock cycle to process the length field
- 1 clock cycle to finalize the generation of the tag
- 5 clock cycles control overhead

With respect to performance, Table I shows that both designs have a very large critical path, leading to a large minimal clock period and thus a performance degradation in comparison to [10]. The authors in [10] report a maximum clock frequency of 324 MHz, while our design reaches 50 MHz for the round-based version and 6 MHz for the unrolled pipelined version. The critical path occurs in the connection of the consecutive cells that are responsible for the loop computations in Algorithm 1.

VI. Conclusion

This work proposes the optimal use of DSP slices and BRAM tiles in UltraScale+ FPGAs for the construction of AES-GCM architectures. While similar work has already been done for the AES core, this work also optimizes the GF(2^{128}) multiplier in the GCM operation. Further, it improves on previous efforts of mapping AES on DSP logic and BRAMs. Two architectures are implemented and evaluated: a round-based architecture, performing both AES and the multiplication in 10 cycles, and a fully unrolled pipelined architecture, executing AES as well as the multiplication in 1 cycle.

The implementation results show that we manage to reduce the occupied LUTs and FFs by a factor 20 in comparison to the smallest known AES-GCM implementation on FPGA (to our knowledge). This is thanks to the optimal use of the DSP slices and the BRAM tiles. However, the use of the DSP slices results in a relatively long critical path in the GF(2^{128}) multiplier, leading to a significant performance degradation in comparison to related work. We can therefore conclude that the architectures proposed in this work are mainly interesting to be added as IP cores to FPGA applications that already occupy many LUTs and FFs, but have a lot of free DSP slices and BRAM tiles.

VII. Acknowledgement

This work was partially funded by the DRASTIC project (CELSA/17/033).

References

[1] S. Drimer, T. Güneysu, and C. Paar, "DSPs, BRAMs and a Pinch of Logic: New Recipes for AES on FPGAs," in *FCCM*, pp. 99–108, IEEE, 2008.

[2] (NIST), "Advanced Encryption Standard (AES) ," Tech. Rep. FIPS-197, U.S. Department of Commerce, 2001.

[3] (NIST), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," Tech. Rep. SP800-38D, U.S. Department of Commerce, 2007.

[4] Xilinx, "UltraScale Architecture DSP Slice User Guide UG473," 2019.

[5] J. Daemen and V. Rijmen, *The Design of Rijndael*. Berlin, Heidelberg: Springer-Verlag, 2002.

[6] T. Güneysu and C. Paar, "Ultra High Performance ECC over NIST Primes on Commercial FPGAs," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 62–78, Springer, 2008.

[7] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, 2014.

[8] R. Shahid, M. U. Sharif, M. Rogawski, and K. Gaj, "Use of Embedded FPGA Resources in Implementations of 14 Round 2 SHA-3 Candidates," in *2011 International Conference on Field-Programmable Technology*, pp. 1–9, IEEE, 2011.

[9] K. Sakiyama, L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede, "Small-footprint ALU for Public-key Processors for Pervasive Security," in *Workshop on RFID Security*, vol. 12, 2006.

[10] G. Zhou, H. Michalik, and L. Hinsenkamp, "Improving Throughput of AES-GCM with Pipelined Karatsuba Multipliers on FPGAs," in *ARC*, pp. 193–203, Springer, 2009.