Benchmarking and Optimizing AES for Lightweight Cryptography on ASICs

Jenny W. Yu and Mark D. Aagaard

University of Waterloo, Waterloo ON, CA
{j99yu,maagaard}@uwaterloo.ca

Abstract. There have been numerous works that focus on optimizing the AES cipher to minimize area. Even with the notion of gate equivalents, the different tools and ASIC libraries used to synthesize the designs makes it hard to objectively compare them. We benchmark and analyze AES-128 encryption cores by implementing and synthesizing them using a set of four ASIC libraries. We show how different implementations of internal AES functions lend themselves better to different architectural options. Using this analysis, we design our own 8-bit AES encryption core, which has a 13% improvement in area and 9% improvement in throughput/area^2 over the next smallest design on STMicro's 65 nm process. It has an area of 1960 GE and a latency of 216 clock cycles.

1 Introduction

Since its conception in 2001, AES has grown into the most popular cipher, finding use in a variety of applications ranging from passive RFID tags, such as the NXP UCODE, to high-performance microprocessors. Although AES is not considered a lightweight cipher, low-area designs have arisen in an effort to make it suitable for small, resource-constrained devices. Research on lightweight implementations of AES is beneficial to embedded systems that use this cipher and to candidates for NIST's lightweight cryptography standardization that are based on it, and provides a context for the benchmarking and analysis of lightweight ciphers. In this paper, we compare 8-bit AES architectures from the works of Mathew [7], Moradi [8], and Hamalainen [5] on four different ASIC libraries to demonstrate how gate equivalent (GE) values can vary substantially even for the same transistor size. From the analysis, we develop our own 8-bit encryption core, Quark-AES. While an 8-bit design may suit some applications, others may need better throughput at the cost of increased area. Thus, we also extend Mathew, Moradi, Hamalainen, and our architecture to 16-bit and 32-bit datawidth designs.

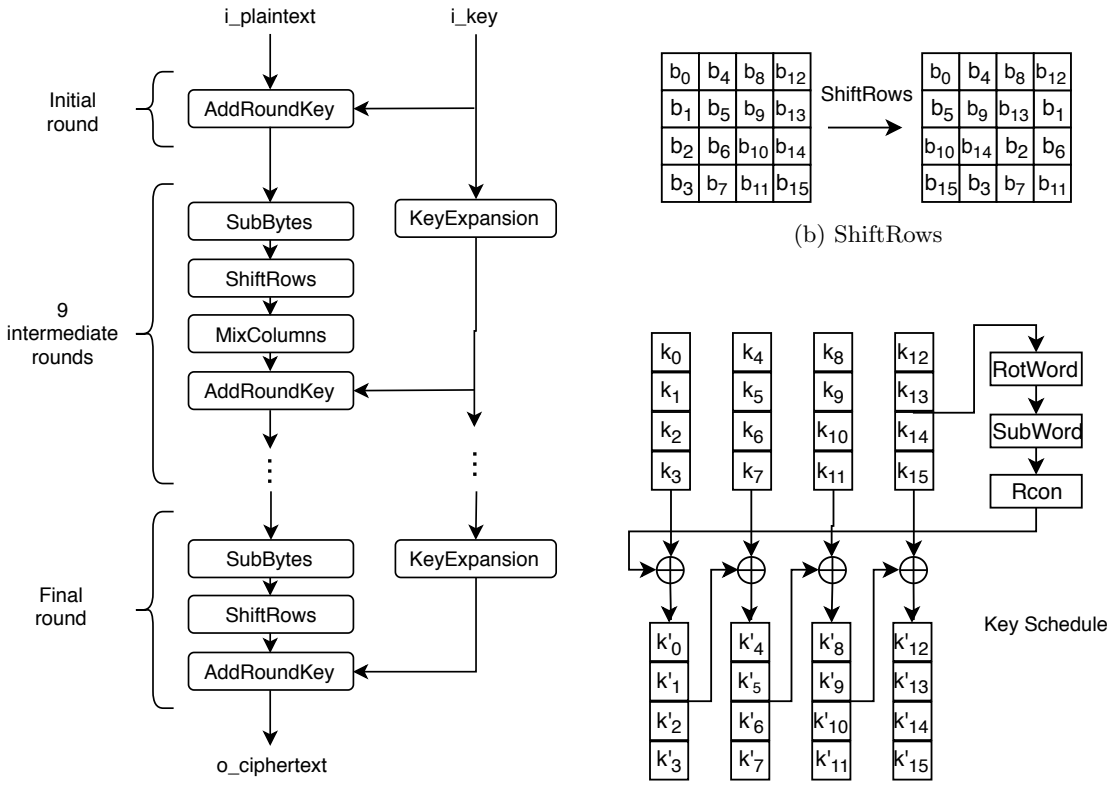
The rest of the paper is organized as follows. Section 2 describes the AES algorithm and Section 3 addresses related works focused on low-area AES designs. The designs that pertain to an encryption-only core are analyzed and benchmarked in Section 4. We present our own low-area encryption core, Quark-AES, in Section 5, which is 13% smaller than the next smallest design. Section 6 summarizes the results of all designs including higher datawidth architectures.

2 The Advanced Encryption Standard Algorithm

The Advanced Encryption Standard (AES) was established by the National Institute of Standards and Technology (NIST) in 2001 [9]. It is a block cipher that operates on block sizes of 128 bits and supports key sizes of 128, 192 and 256 bits. The number of rounds differs depending on the key size. This section describes the AES algorithm for a key size of 128 bits, which requires 10 rounds. The 16 bytes of the message, commonly referred to as the *state*, are typically represented in a 4×4 column-major order matrix.

Each round of AES applies all or a subset of four operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The initial round consists of just AddRoundKey. The nine intermediate rounds apply all the operations, in the order specified above. The final round is the same as an intermediate round except the MixColumns operation is omitted. Each round requires a round key which is derived from the cipher key using the AES key generation algorithm. Figure 1(a) shows a high-level view of the AES-128 algorithm. The following describes the internal operations and the key schedule.

This work was supported in part by the Canadian National Science and Engineering Research Council (NSERC); the Canadian Microelectronics Corp (CMC); and Grant 60NANB16D289 from the U.S. Department of Commerce, National Institute of Standards and Technology (NIST).



(a) High-level AES algorithm

(b) ShiftRows

(c) Key schedule

Fig. 1: AES algorithm

AddRoundKey The round key is XORed with the current state.

SubBytes This layer, the only non-linear operation, applies an S-box function to each byte of the state. The AES S-box is a multiplicative inverse operation in the GF(2^8) field defined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, followed by an affine transformation.

ShiftRows In this step, shown in Figure 1(b), the bytes of the state are permuted. More specifically, the first row of the state remains unchanged, the second row is left rotated by one byte, the third row by two bytes, and the last row by three bytes.

MixColumns This operation transforms each column (four bytes) of the state using a linear transformation. If $a_0a_1a_2a_3$ are four bytes of a column, then the result of a MixColumns operation, $b_0b_1b_2b_3$, is obtained as follows:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2\ 3\ 1\ 1 \\ 1\ 2\ 3\ 1 \\ 1\ 1\ 2\ 3 \\ 3\ 1\ 1\ 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Multiplication is performed in the GF(2^8) field and addition is simply the XOR operation.

Key Generation Each round key is derived from the round key of the previous round (or from the input cipher key, if it is the first round). Suppose the columns from left to right are indexed 0 to 3. First, column 3, $[k_{12}, k_{13}, k_{14}, k_{15}]$, is left rotated by one byte (RotWord): $[k_{13}, k_{14}, k_{15}, k_{12}]$. Then, each byte of the result undergoes the Sbox function (SubWord) and the first byte gets XORed with the round constant, *rcon*, which is generated as

$$rcon_i = \begin{cases} 1 & \text{if } i = 1 \\ 2 \cdot rcon_{i-1} & \text{if } i > 1 \text{ and } rcon_{i-1} < \text{0x80} \\ (2 \cdot rcon_{i-1}) \oplus \text{0x1B} & \text{if } i > 1 \text{ and } rcon_{i-1} \geq \text{0x80} \end{cases}$$

Column 0 of the new key, $[k'_0, k'_1, k'_2, k'_3]$, is obtained by XORing each byte of column 0 with each byte of the modified column 3. Columns $i = 1, 2, 3$ of the new key are then obtained by performing the XOR of column $i - 1$ of the new key and column i of the current key. The key schedule is illustrated in Figure 1(c).

3 Related Work

Low-area implementations of AES have appeared since the outset. In the same year that AES became standardized, Satoh *et al* [10] introduced a compact architecture that combined the encryption and decryption datapaths. They achieved an area of 5400 GE on a 0.11 um process and a latency of 54 clock cycles by using a datawidth of 32 bits. One of the first 8-bit architectures of AES came from Feldhofer *et al* [4]. They implemented both encryption and decryption cores on a 0.35 um process, which together incurred 4400 GE. Encryption takes 1032 clock cycles while decryption requires 1165 cycles. The authors use a single S-box for both SubBytes and key generation. Hamalainen *et al* [5] also designed an 8-bit encryption architecture, which occupies an area of 3100 GE on a 0.13 um process. Performing on-the-fly key schedule and encryption in parallel, their design achieves a latency of 160 clock cycles. The authors use two instances of the S-box (one for encryption, one for key schedule), which borrows Canright's design [3]. Their ShiftRow operation is performed serially using Jarvinen's Byte Permutation Unit [6], which outputs the bytes of the state in ShiftRows order.

Moradi *et al* [8] took a holistic approach for minimizing the total area, rather than optimizing each component individually. Their 8-bit encryption module achieves an area of 2400 GE on a 0.18 um process and has a latency of 226 clock cycles. Moradi uses a single S-box that is shared between encryption and key generation and has nearly 100% utilization. Building on the work of Moradi, Banik *et al* [2] developed Atomic-AES and Atomic-AES v2.0, which perform both encryption and decryption. The former has a 226 clock cycle latency for both encryption and decryption and an area of 2605 GE on STM 90 nm CMOS library. Atomic-AES v2.0 achieves a smaller area (2060 GE) at the cost of greater latency (246/326 for encryption/decryption, respectively).

A sub-2000 GE implementation of an AES encryption module appeared in the work of Mathew *et al* [7]. Occupying 1947 GE on a 22 nm process, the design performs encryption in 336 clock cycles. The 8-bit architecture performs encryption entirely in a new composite field, applying the isomorphic mapping prior to the first round and then the inverse mapping at the end of all AES rounds. One S-box is used for both encryption and key generation, which occur in alternation. The authors did an exhaustive test of all possible polynomials for the ground-field and extension-field and chose the ones that provided the smallest area. Their results show that the optimal encrypt and decrypt cores use different polynomials. Sub-atomic AES, designed by Wamser *et al* [11], is the smallest encryption/decryption dual core to date, reported to be more than a 10% reduction in area compared to Atomic-AES v2.0. However, it comes at a cost of a large increase in latency: 689/1281 clock cycles for encryption/decryption, respectively.

4 Benchmarking and Analyzing Existing Architectures

The architectural and component options for an 8-bit design are ample. The bytes can be internally processed in a row-major or column-major fashion. AES operations can be implemented serially or non-serially. Encryption and key expansion can occur in parallel, requiring duplicate hardware, or serially, saving area at the cost of increased latency. AES operations can also take on a different order as long as dependencies are maintained (*e.g.*the relative order of ShiftRows and SubBytes does not matter). This section offers some lessons learned from our iterations and variations of existing works, provides an analysis of the different implementations of AES components, and demonstrates which component implementations lend themselves better to which kinds of architectures. We use Section 4.1 to introduce the different component options and Section 4.2 to examine them more closely.

Of the aforementioned works, Moradi, Mathew, and Hamalainen's designs stand out for their area and time efficiency. Mathew's work reports the smallest encryption-only area, Moradi's work achieves nearly the lowest possible cycle count for a single S-box design, and Hamalainen's work

implements a serialized method of the ShiftRows operation. To better compare these designs, we implemented and synthesized them on a common technology. Logic synthesis was performed with Synopsys Design Compiler version P-2019.03 using the compile_ultra command and clock gating. Physical synthesis (place and route) and power analysis were done with Cadence Encounter v14.13 using a density of 95%. We used Mentor Graphics ModelSim SE v10.5c for simulations. All area results are post place-and-route and power analysis is based on timing simulation. The ASIC cell libraries used were ST Microelectronics 65 nm CORE65LPLVT at 1.25V, TSMC 65 nm at tpfn65gpgv2od3_200c and tcbn65gplus_200a at 1.0V, ST Microelectronics 90 nm CORE90GPLVT and CORX90GPLVT at 1.0V, and IBM 130 nm CMRF8SF LPVT with SAGE-X v2.0 standard cells. Some past works have used scan-cell flip-flops to reduce area because these cells include a 2:1 multiplexer in the flip-flop, which incurs less area than using a separate multiplexer. We chose not to use scan-cell flip-flops because their use as part of the design would prevent their insertion for fault-detection and hence, prevent the circuit from being tested for manufacturing faults.

4.1 Architecture Comparison

It is important to note that our analysis is on the architectures of Mathew's, Moradi's, and Hamalainen's works, thus our implementations follow their descriptions and circuits as closely as possible, with the exception of common components which remain uniform across all designs. For instance, we use the same S-box (Mathew's) in all three architectures. Since Moradi's and Hamalainen's design are done in the original AES field, we put the isomorphic mappings immediately before and after the S-box. To allow rapid prototyping of many different architectures, we use a binary encoding for our counters. Once an architecture is finalized, the implementation could be optimized by replacing the binary counter with an LFSR. In this section, we implemented the designs using STMicro 65 nm process.

Mathew's Architecture To our knowledge, the smallest reported encryption-only module of AES is from the work of Mathew, who reported an area of 1947 GE on a 22 nm process. Mathew applies their custom composite field to the entire algorithm and performs almost all AES operations in the new basis. The design uses one instance of the S-box and has a total latency of 336 clock cycles. In each round, 16 clock cycles are dedicated to encryption and 16 clock cycles to key expansion. During encryption, AddRoundKey, SubBytes, and MixColumns are performed serially, and the results are stored in 128 *intermediate* registers. The ShiftRows operation is performed in the first clock cycle of key expansion mode when the contents of the intermediate register are transferred to the state register. Because AddRoundKey, SubBytes, and MixColumns are applied in every clock cycle and MixColumns depends on ShiftRows, the ShiftRows operation is moved to the beginning of the round. This has two consequences: the input bytes must be loaded in ShiftRows order and the key must be added to the state in ShiftRows order by introducing a 4-to-1 multiplexer. The authors also use scan flip-flops in their design.

Analysis of Mathew's Architecture Our implementation of Mathew's design achieves an area of 2640 GE. The large discrepancy between our and Mathew's area could be due to two reasons. First, we opted not to use scan flip flops. Since scan flip flops incur less area than combination of a multiplexer and a flip-flop, our implementation will be larger. Second, while the composite field selected by Mathew may be optimal on their technology, it might not be on ours.

The main drawbacks of Mathew's design are the additional intermediate registers, the need to reorder inputs, and the 336 clock cycle latency. Most of the intermediate registers can be eliminated simply by writing the result of MixColumns directly into the data registers instead of into the intermediate registers. This eliminates 12 of the 16 8-bit registers (4 are still needed for temporary storage of the result) and introduces four 8-bit multiplexers. This version achieves 2240 GE in our experiments. Another option for MixColumns is to have a 32-bit implementation that operates on a column at a time and to do it in 4 clock cycles after the ShiftRows operation. The advantage of this is the elimination of all intermediate registers and no longer needing to reorder inputs. The area result for this implementation is 2270 GE.

While Mathew decided it would be better to do the entire algorithm in the composite field, Canright argues that this may be less efficient than having just the S-box in a composite field because the simplicity of the constants in the MixColumns operation in the original basis, 0x03 and 0x02, would be lost in the composite field [3]. As an experiment, we implemented a version that performs the isomorphic mapping immediately before and after the S-box, leaving all other operations in the original basis. The results showed a decrease in area from 2640 GE to 2580 GE.

Moradi's Architecture Moradi's 8-bit encryption architecture is next smallest at a reported 2400 GE and stood out for its low latency for a single S-box design. Sixteen clock cycles are dedicated to AddRoundkey and SubBytes, one to ShiftRows, and four to the simultaneous execution of MixColumns and SubWord (of the key expansion), for a total round latency of 21 clock cycles. The S-box borrows Canright's design and MixColumns is performed on a column at a time, requiring no additional registers. Moradi's design also uses scan flip-flops and requires both inputs and outputs to be reordered, as bytes are processed in a row-major order.

Analysis of Moradi's Architecture Our implementation of the design occupies 2370 GE. We also point out that our implementation uses a larger S-box (the same one we used in our implementation of Mathew's design) and a regular binary counter (instead of an LFSR).

A slight improvement on latency can be made by doing the ShiftRows operation in the same clock cycle as the final AddRoundKey byte. This eliminates one clock cycle in each round, resulting in a total cycle count of 216 compared to the original 226. Although this causes the location of the muxes to change, the number of muxes remains the same (every register in the second and third rows require a mux and the left-most register of the bottom row requires a mux). In our experiments, making this change caused area to decrease by 20 GE, which could be attributed to how the tools optimize the design.

Other improvements have been suggested by Banik *et al* [2], who extended Moradi's work to support both encryption and decryption. The registers in the two middle columns of the key do not need to shift during clock cycles 17-20; thus, they do not need to be scan flip-flops or, in our case, require muxes. In Atomic-AES v2.0, ShiftRows is performed over 3 clock cycles, during which the rows are selectively shifted by one byte at a time. This eliminates all muxes except the ones for the rightmost column. More details appear in Section 4.2.

Moradi's paper states that using a row-major design reduces area by 13.5% and that if column-wise ordering is needed, 20 additional 8-bit wide 2-to-1 multiplexers are required. However, in our experiments, the column-major design in fact showed a decrease in area: 2280 GE. According to our analysis, a column-major design should decrease area because although more muxes are required for the ShiftRows operation, a greater number of muxes are saved in the key register. In a row-major design, 18 muxes (9 for ShiftRows and 9 for key register) are required. In a column-major design, 15 muxes are needed (12 for ShiftRows and 3 in key). See Section 4.2 for details. Moradi argues that the extra registers and control logic associated with a serial version of MixColumns would exceed the area of a combinational 32-bit MixColumns. Our analysis is presented in Section 4.2.

Hamalainen's Architecture At 3100 GE, Hamalainen's architecture uses two instances of the S-box, one for encryption and the other for the key schedule. The authors did the ShiftRows operation serially by employing the Byte Permutation Unit (BPU) developed by Jarvinen *et al* [6]. Consisting of 12 8-bit state registers and some muxes, the BPU outputs the bytes in the correct order by reading from one of four registers (depending on the clock cycle) and systematically reordering the bytes as they shift through the state registers. Most of the BPU registers simply maintain their normal shifting operation as a shift register and do not require muxes to preface them. MixColumns is also performed serially, similar to the way Mathew did, using four additional 8-bit registers to store the accumulating result. The SubBytes operation is performed between ShiftRows and MixColumns operations, which differs slightly from the AES specification, but this rearrangement has no effect on functional correctness.

Analysis of Hamalainen's Architecture Our implementation of the design achieves 2260 GE. Using the BPU has two benefits. First, the area overhead is small. While in the straightforward implementation, the ShiftRows operation requires twelve 8-bit 2-to-1 muxes, the BPU requires only three 8-bit 2-to-1 muxes and one 8-bit 4-to-1 mux (equivalent to three 2-to-1 muxes). Second, now that every operation is serialized, every clock cycle is uniform, making the control logic very simple and reducing the number of muxes. As a result, the round latency can be kept to 16 clock cycles, which is the minimum latency of an 8-bit design, since there are 16 bytes of state.

Summary Table 1 shows a summary of the discussed designs. Latency is measured from the clock cycle the first byte goes in until the clock cycle the last byte becomes ready (*i.e.* it includes cycles for loading and unloading). To allow rapic prototyping of many designs, we decided not to implement simultaneous loading and unloading in our designs because the overhead would be similar for all of the

designs we evaluated. While Moradi and Hamalainen report both the latency including loading/unloading and the effective latency when load/unload is performed simultaneously, Mathew reports only one latency value of 336, which appears to be the effective latency. To keep the comparisons consistent, we add 16 clock cycles to Mathew's reported latency to account for loading.

Table 1: Benchmarking on ST 65 nm process

Design	**Notes**	**Area** (GE)	**Latency** (clk cycles)	**Power** (mW)	**Energy** (nJ/bit)
Mathew [7]		2640	352	0.109	29.9
Mathew variation 1	most intermediate registers removed	2240	352	0.122	33.6
Mathew variation 2	no external reorder of inputs	2270	352	0.090	24.6
Mathew variation 3	original AES basis	2580	352	0.110	30.1
Moradi [8]		2370	226	0.101	17.9
Moradi variation 1	latency reduced	2350	216	0.100	16.9
Moradi variation 2	column-major	2280	226	0.097	17.1
Hamalainen [5]		2260	176	0.145	20.0

4.2 Taxonomy of Design Choices

In this section, we compare different ways of implementing MixColumns, ShiftRows, and Key Expansion. Since SubBytes and AddRoundKey are byte-wise operations, they lend themselves well to serial implementations and do not have many architectural options.

Mix Columns There are three methods of implementing MixColumns: 1. non-serialized, 2. serial with four 8-bit registers, and 3. lightweight serial with two 8-bit registers. The non-serialized method is simply the straightforward method of implementing MixColumns for a single column. It can be done combinationally without the need for any registers. Hamalainen and Mathew both opted for a serialized version, which requires four 8-bit registers to store the temporary result. A third method appeared in the work of Wamser [11], who implemented a serialized version of MixColumns requiring only two additional registers, based on the work of Ahmed *et al* [1]. Ahmed shows that, if $[s_0, s_1, s_2, s_3]$ is a column, then the MixColumns result, $[s'_0, s'_1, s'_2, s'_3]$ can be calculated as

$$\begin{aligned} tmp &= s_0 \oplus s_1 \oplus s_2 \oplus s_3 \\ s'_0 &= s_0 \oplus tmp \oplus [2 \times (s_0 \oplus s_1)] \\ s'_1 &= s_1 \oplus tmp \oplus [2 \times (s_1 \oplus s_2)] \\ s'_2 &= s_2 \oplus tmp \oplus [2 \times (s_2 \oplus s_3)] \\ s'_3 &= s_3 \oplus tmp \oplus [2 \times (s_3 \oplus s_0)] \end{aligned}$$

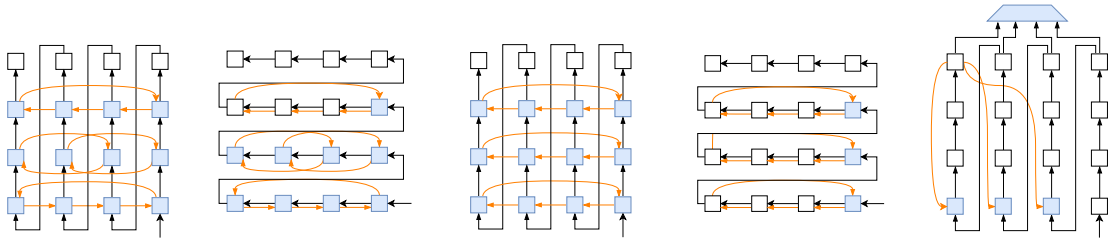
In this method, only two values, tmp and s_0, need to be stored in registers. Each MixColumns byte is calculated and ready in one clock cycle, rather than accumulated over four clock cycles. We also test the effect of implementing the methods in Mathew's composite field. Table 2 lists the register count and area. Results show that the lightweight serial method in the original basis achieves the lowest area

Table 2: MixColumns implementations

Method	**Basis**	**Num. of registers** (8-bit)	**Area** (GE)
Non-serial	original	0	226
	Mathew's	0	366
Serial	original	4	267
	Mathew's	4	303
Serial lightweight	original	2	202
	Mathew's	2	234

and the non-serial method in Mathew's composite field gives the highest area. For a given method, the original basis always produces lower area than Mathew's basis. In the original basis, the non-serial method obtains lower area than the serial method, but in the composite field, the non-serial method obtains greater area than the serial method. This indicates that the best method depends on the area trade-off between the combinational circuitry, which is determined by the basis, and the registers. Since Moradi stays in the original basis, the non-serial method is the better option. In Mathew's basis, the combinational circuitry of the non-serial method appears to be larger than the area of the four 8-bit registers, thus the serial method is the better option.

ShiftRows We saw three ways of implementing ShiftRows: 1. straightforward method in one clock cycle, 2. over 3 clock cycles, and 3. serially using Jarvinen's BPU. Since the ShiftRows operation is a row-wise operation, a row-major design can offer efficiencies which a column-major design cannot. In a column-major implementation, the single clock cycle method requires 12 muxes (every register in the last 3 rows of the matrix needs one), shown in Figure 2(a). However, in a row-major implementation,



(a) Column-major one clock cycle (b) Row-major one clock cycle (c) Column-major three clock cycle (d) Row-major three clock cycle (e) BPU

Fig. 2: ShiftRow implementations. Registers are indicated by square boxes. Black arrows indicate regular shifting operation while orange arrows indicate ShiftRows shifting. Blue registers need to be prefaced with a mux.

the ShiftRows operation for the second row of the matrix (*i.e.* left shift by one byte) happens to be the same operation as the shift register. Thus, the ShiftRows operation shown in Figure 2(b) requires one mux for every register in the third and fourth rows and only one mux in the second row, for a total of nine muxes. Even more registers can be saved if ShiftRows is done over three clock cycles. In each clock cycle, the second, third, and fourth rows are selectively shifted and they shift by only one byte at a time. Figure 2(d) demonstrates that this method requires only three 8-bit muxes. The drawback is the increase in latency. In a column-major design, performing ShiftRows over three clock cycles doesn't offer any benefits because the column-major shifting doesn't coincide with the row-wise shifting of ShiftRows shifting, shown in Figure 2(c). Thus, it still needs 12 muxes. The final method of ShiftRows, the BPU, is designed for a column-major ordering. It requires three 2:1 muxes and one 4:1 mux. If we count the 4:1 mux as three 2:1 muxes, then the total mux count is six for BPU, depicted in Figure 2(e). Table 3 lists the different methods and their corresponding mux count.

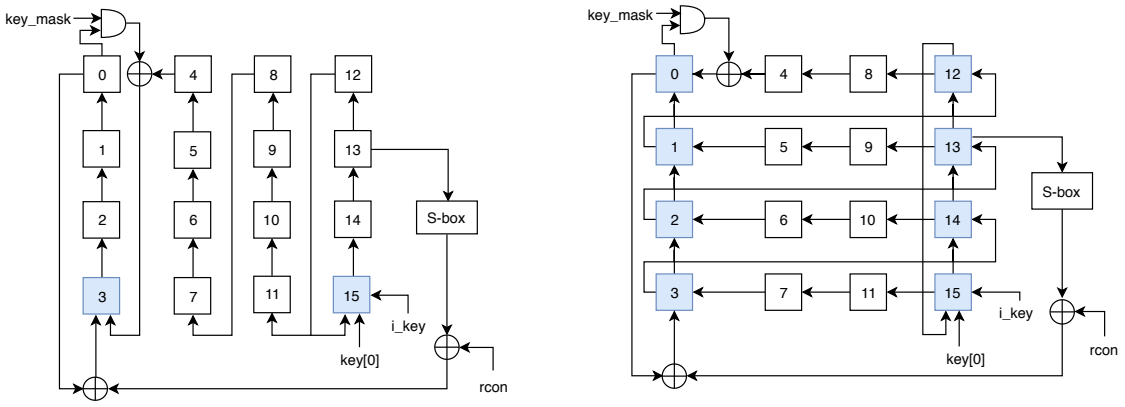
Table 3: ShiftRows Implementations

Method	**Column- or row-major design?**	**Number of 2:1 muxes**
One clock cycle	column	12
	row	9
Three clock cycle	column	12
	row	3
BPU	column	6

Key Expansion The movement of the key schedule should be compatible with the movement of the state register. That is, if the state register shifts in a row-major order, then the key register should shift in row-major order as well, so that the AddRoundKey operation can be done efficiently. If the key and state registers differ in order, then additional circuitry is needed to make them compatible,

which adds area. For instance, in Mathew's circuit, the state register is in ShiftRows order when it is time to do AddRoundKey. Thus, they must use a 4:1 mux to select the correct byte from the key register to add to the state.

While a row-major design can offer area efficiency in the state register (as we saw in the ShiftRows discussion), it complicates the circuitry surrounding the key register. We will show how by using Moradi's key schedule architecture and a corresponding column-major version of it. The key algorithm naturally lends itself better to a column-major design. The left-most column of the next round key is derived from a transformed version of the right-most column. As we populate the left column, we can either rotate the right-most column and keep reading from register 13 or keep the column static and use a 4:1 mux. In a column-major design, keeping the column rotating is a better choice because it follows the normal column-major shifting operation. Muxes need to preface the bottom registers of the rightmost and leftmost columns, shown in blue in Figure 3(a). In a row-major design, the column-wise shifting that is required to generate the first column forces a mux to be used on each register in the left-most and right-most columns, shown in Figure 3(b).



(a) Column-major design requires 3 muxes

(b) Row-major design requires 9 muxes

Fig. 3: Key schedule implemented in column-major and row-major style. Blue registers require muxes. The number of muxes needed for a register equals the number of arrows going into the register minus one.

4.3 Summary

Different design goals will motivate different design choices and the analysis above can facilitate the decision process. For example, if reordering of inputs/outputs is acceptable, then a row-major design could be the best choice. Given a row-major design, the ShiftRows method that offers the lowest area would be doing it over three clock cycles. On the other hand, if a column-major design is favoured, then ShiftRows could be implemented as the BPU, the lowest-area method in a column-major design.

5 Our Design: Quark-AES

We extract the best features of all the aforementioned designs to realize a novel design. We wanted a single instance of the S-box, the minimum latency for a one-S-box design, the lightweight MixColumns implementation, the byte permutation unit, and a column-major design so that inputs and outputs do not have to be reordered externally. The challenge is assembling the desired components into a low-area architecture while adding as few registers and muxes as possible. Although every AES component we chose is serial, we cannot simply stack all of them together in a combinational path. The lightweight MixColumns algorithms requires four consecutive bytes of the ShiftRows output to be available at the beginning of every four clock cycles. Because ShiftRows is a serial operation, the MixColumns unit would require four extra registers to store these bytes. To achieve this without the use of any additional registers, we delayed the MixColumns operation by four clock cycles so we can use four state registers for storage. With careful placement of components, all 16 state registers, which is exactly enough, can be used for the 12 registers in the BPU and the 4 in MixColumns.

5.1 Architecture

In our architecture, every encryption function (AddRoundkey, SubBytes, ShiftRows, MixColumns) is performed serially, requiring only 16 clock cycles in a single AES round. The remaining four clock cycles in the round are devoted to the SubWord operation in the key schedule, during which the state register remains idle. Therefore, the S-box has 100% utilization. The architecture is illustrated in Figure 4. To offer a fair comparison of our design against existing works, we kept the S-box the same as the one used in our analysis in Section 4, which is Mathew's S-box.

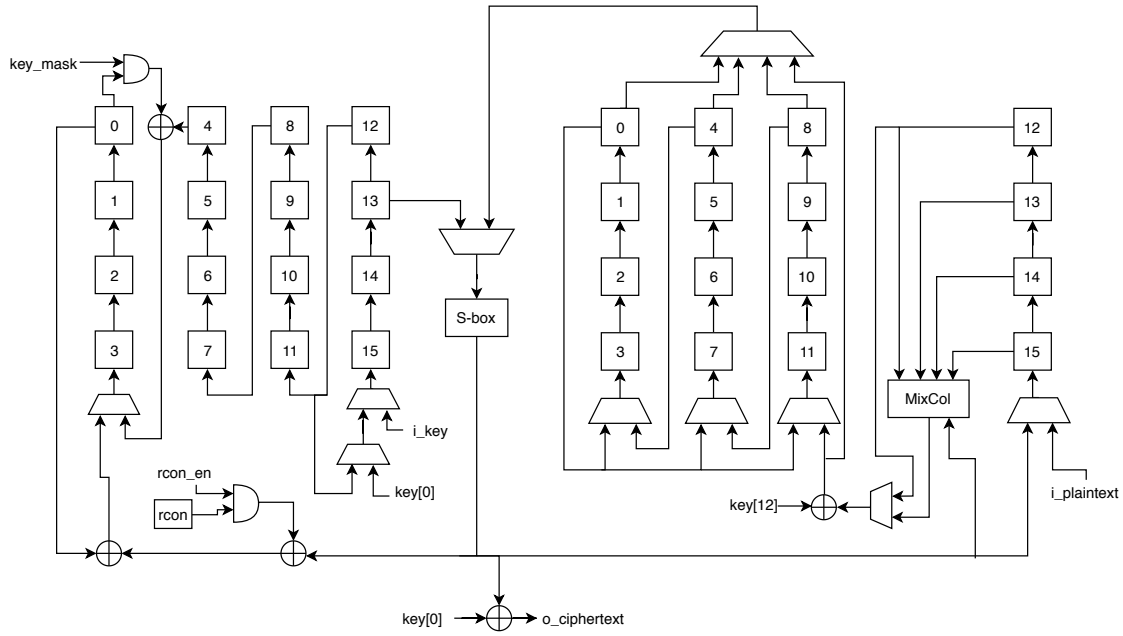
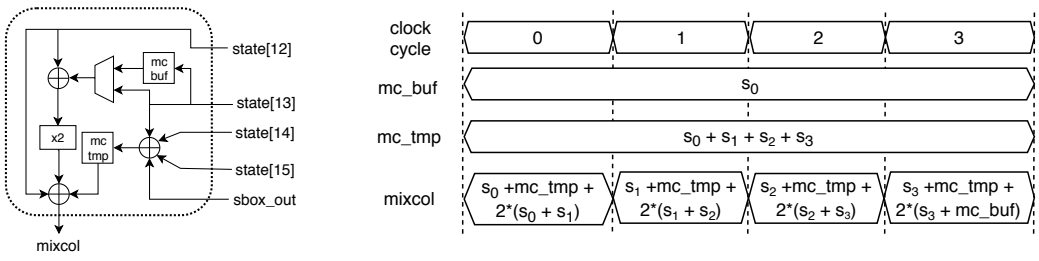


Fig. 4: Quark-AESarchitecture with key register on the left and state register on the right

ShiftRows The ShiftRows operation is implemented as the BPU, which consists of state registers [0..11], the three 2-to-1 muxes, and the 4-to-1 mux. Refer to [6] for details of the BPU operation.

Mix Columns We employ the lightweight implementation of MixColumns (the third method in Table 2), depicted in Figure 5. Because the values tmp and s_0 are required in clock cycles after they become available, their values must be saved to registers. Every four clock cycles, the sum tmp is stored in register mc_tmp and value s_0 is stored in register mc_buf. The MixColumns operation requires four consecutive bytes to be available after the ShiftRows operation (to calculate tmp). Since ShiftRows is a serial operation, it takes four clock cycles for four bytes to be ready. Thus, we delay the MixColumns operation by four clock cycles (until state[12..15] is populated) and place the circuitry after the state[12] register.



(a) Circuit (b) Dataflow

Fig. 5: Quark-AES MixColumns circuit and operation

Key Expansion The key generation for the next round begins in clock cycle 16 of the current round. For four clock cycles, the left-most and right-most columns rotate by one byte to generate the left-most column for the next round key. Once the first column is generated, subsequent bytes are generated by doing (key[0] $\oplus$ key[4]), the result of which is written to key[3]. The signal `key_mask` is asserted for 12 clock cycles after the first column of the next round key is generated. By clock cycle 12, all key bytes have been generated, so `key_mask` is set to 0 and the key register needs only to shift the bytes out.

Control There are two counters in our design: one for counting the clock cycles within a round and another to keep track of the round.

5.2 Dataflow

Clock cycles 0 to 15 ShiftRows, SubBytes, MixColumns, and AddRoundKey are all performed serially. MixColumns and AddRoundkey are delayed by four clock cycles (relative to the start of the round), starting in clock cycle 4 and extending 4 clock cycles into the next round, shown in Figure 6. Shiftrows, requiring only 12 clock cycles, occurs in clock cycles 0 to 11. After clock cycle 11, all the bytes are in the correct order and need only to be shifted out. In the first round, we bypass the MixColumns circuitry and the AddRoundKey takes state[12] instead of MixColumns output.

Clock cycles 16 to 19 The state register remains idle during these clock cycles. The S-box is used to calculate the first four bytes of the next round key, which get stored in the left-most column of the *key* register. During these four clock cycles, the left-most and right-most columns of the key register rotate, while the rest of the key registers remain idle.

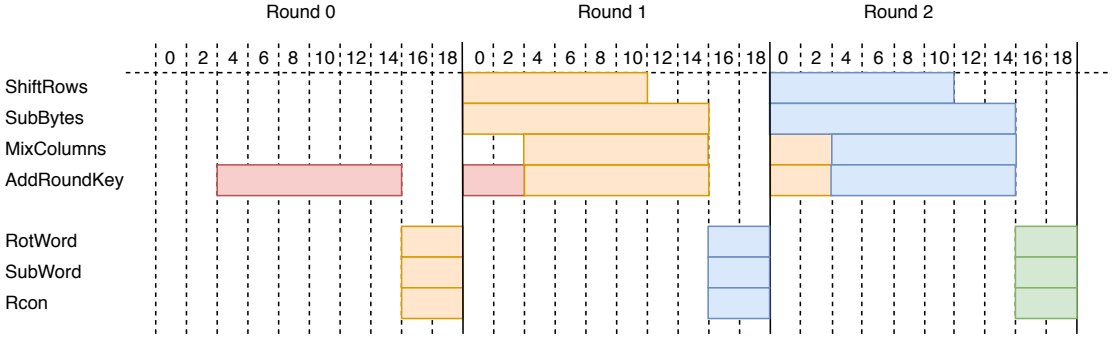


Fig. 6: Dataflow of Quark-AES showing first 3 rounds. Blocks of the same colour represent operations belonging to the same AES round.

6 Results

While an 8-bit datawidth may be an appealing design for some applications, other applications could prioritize throughput over area and desire a higher datawidth. We explored these options and implemented 16-bit and 32-bit datawidth versions of Mathew, Moradi, Hamalainen, and Quark-AES. The results are summarized in Table 4. The last two rows include the area results for Atomic-AES and Atomic-AES v2.0, which we synthesized using their publicly available source code. Although they are dual-featured cores supporting both encryption and decryption, it is nice to see where they stand in comparison to encryption-only architectures. The latency values include cycles required for the loading and unloading, and the throughput values do not include simultaneous loading/unloading of the data. The last two columns are optimality metrics of $throughput/area$ and $throughput/area^2$. The first metric is useful if throughput and area are equally important, while the second metric takes power into account by using area as its approximation. For convenience, we reiterate Mathew, Moradi, and Hamalainen's reported area results: 1947, 2400, and 3100 GE, respectively. At 1960 GE, Quark-AES has the smallest area of all the designs. It is 13% smaller than the next smallest design (Hamalainen's) when synthesized on ST 65 nm. In real-world applications, it is often the case that data may not be ready or available every clock cycle. To be practical, our design supports bubbles during loading. The 8-bit Quark-AES outperforms other 8-bit designs in power, energy, and T/A^2.

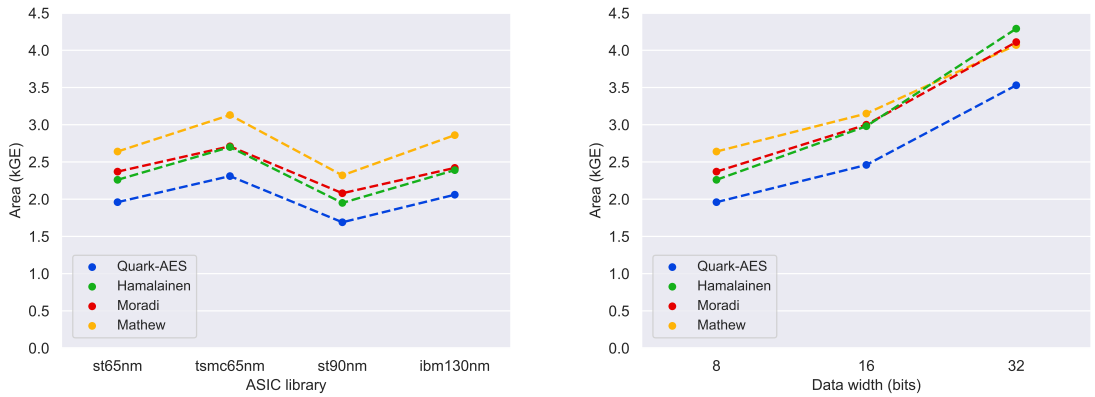
Table 4: Summary of designs on ST 65 nm process

Design	**Data width** (bits)	**Num. S-boxes**	**Area** (GE)	**Latency** (clk cycles)	**Through-put** (bits/clkcycle)	**Power** (mW)	**Energy** (nJ/bit)	**T/A** ($10e-6$)	**T/A^2** ($10e-8$)
Mathew	8	1	2640	352	0.364	0.109	29.9	138	5.22
	16	2	3150	176	0.727	0.174	23.9	231	7.33
	32	4	4070	88	1.455	0.238	16.4	357	8.78
Moradi	8	1	2370	226	0.566	0.101	17.9	239	10.1
	16	2	3000	118	1.085	0.130	11.9	362	12.1
	32	4	4110	64	2.000	0.191	9.54	487	11.8
Hamalainen	8	2	2260	**176**	**0.727**	0.145	20.0	**322**	14.2
	16	4	2980	**88**	**1.455**	0.200	13.7	**488**	16.4
	32	8	4290	**44**	**2.909**	0.269	9.24	**678**	15.8
Quark-AES	8	1	**1960**	216	0.593	**0.091**	**15.3**	302	**15.4**
	16	2	**2420**	108	1.185	**0.129**	**10.9**	482	**19.6**
	32	4	**3530**	54	2.370	**0.172**	**7.25**	671	**19.0**
Atomic-AES	8	1	2680	226	0.566				
Atomic-AES v2.0	8	1	2480	246	0.520				

It is difficult to compare works that use different ASIC libraries and toolchains. Even though the gate equivalent metric attempts to normalize across different technologies, there are still considerable variations among different libraries, illustrated in Table 5. Flop_ce denotes chip-enabled flip-flop. Figure 7(a) demonstrates area variations of the AES designs on four different ASIC libraries. The relative rank of Hamalainen's and Moradi's area results differs depending on the ASIC library used. Figure 7(b) shows how well the architectures extend to higher datawidths (16-bit and 32-bit). While Mathew's design has the highest area at 8 bits, the 32-bit version is smaller than Moradi's and Hamalainen's.

Table 5: Area of common cells for different libraries

	NAND		**AND**		**MUX**		**XOR**		**flop**		**flop_ce**	
	μm^2	GE	μm^2	GE	μm^2	GE	μm^2	GE	μm^2	GE	μm^2	GE
st65nm	2.08	1.00	2.60	1.25	4.16	2.00	4.16	2.00	7.80	3.75	10.40	5.00
tsmc65nm	1.44	1.00	2.16	1.50	3.24	2.25	3.60	2.50	6.84	4.75	9.36	6.50
st90nm	4.39	1.00	5.49	1.25	8.78	2.00	7.68	1.75	14.27	3.25	19.76	4.50
ibm130nm	5.76	1.00	7.20	1.25	12.96	2.25	11.52	2.00	41.76	7.25	34.56	6.00



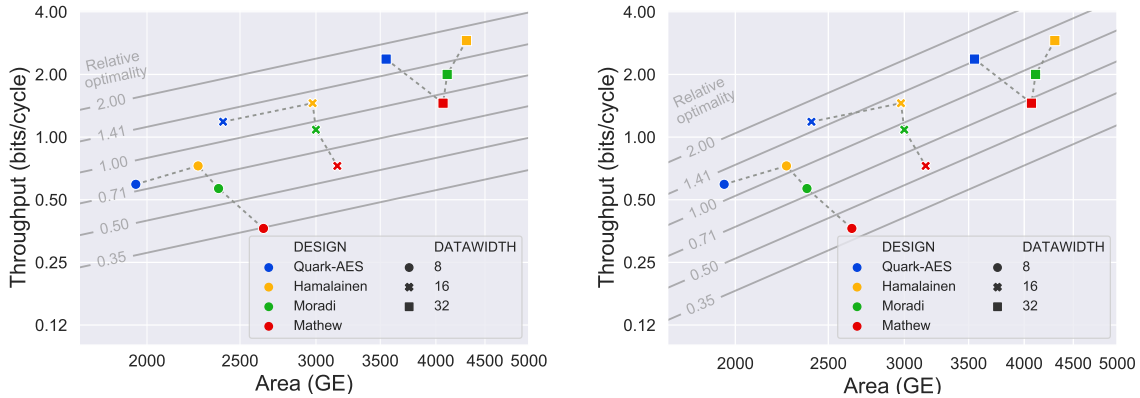
(a) Area on four different ASIC libraries

(b) Area vs data width

Fig. 7: Area trends

We also present a comparison of the designs in terms of two optimality measures, $\frac{\text{throughput}}{\text{area}}$ and $\frac{\text{throughput}}{\text{area}^2}$, illustrated in Figure 8. Data points of the same colour have the same architecture and the marker style indicates the datawidth in bits. In both graphs, the y axis is *log* scaled. The x axis in Figure 8(a) is scaled as $log(x)$ while the x axis in Figure 8(b) is scaled as $log(x^2)$. The grey contour lines

represent normalized optimality values: $\frac{\text{optimality}}{\text{average optimality}}$. The average optimality of all 12 datapoints is represented by the contour line labelled 1.00. Quark-AES consistently offers the lowest area in each datawidth category. In Figure 8(a), its optimality is slightly below Hamalainen's for the 8-bit architecture, but on par for the 16-bit and 32-bit designs. In Figure 8(b), Quark-AES outperforms (*i.e.* has higher optimality than) every other architecture in each datawidth category.



(a) Throughput/area (b) Throughput/area^2

Fig. 8: Optimality

7 Conclusion

In this paper, we provided an analysis of several architectural and component options for 8-bit AES designs. Based on this analysis, we were able to create a novel architecture, Quark-AES, that offers the benefits of low latency (216 clock cycles), single S-box instance, and no requirement to reorder inputs or outputs externally. It achieves an area of 1960 GE, which is 13% smaller than the next smallest encryption core synthesized using our toolchain. In the future, we plan to extend the design to support both encryption and decryption and add micro-optimizations.

References

1. Ahmed, E.G., Shaaban, E., Hashem, M.: Lightweight Mix Columns implementation for AES. In: Proceedings of the 11th WSEAS International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering. pp. 48–53. MMACTEE'09 (2009)
2. Banik, S., Bogdanov, A., Regazzoni, F.: Atomic-AES: A compact implementation of the AES encryption/decryption core. vol. 10095, pp. 173–190 (12 2016)
3. Canright, D.: A very compact S-Box for AES. In: Rao, J.R., Sunar, B. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2005. pp. 441–455 (2005)
4. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. IEE Proceedings - Information Security **152**(1), 13–20 (Oct 2005)
5. Hämäläinen, P., Alho, T., Hännikäinen, M., Hämäläinen, T.D.: Design and implementation of low-area and low-power AES encryption hardware core. In: 9th EUROMICRO Conference on Digital System Design (DSD'06). pp. 577–583 (Aug 2006). https://doi.org/10.1109/DSD.2006.40
6. Järvinen, T., Salmela, P., Hämäläinen, P., Takala, J.: Efficient byte permutation realizations for compact AES implementations. In: 2005 13th European Signal Processing Conference. pp. 1–4 (Sep 2005)
7. Mathew, S., Satpathy, S., Suresh, V., Kaul, H., Anders, M., Chen, G., Agarwal, A., Hsu, S., Krishnamurthy, R.: 340mv1.1v, 289gbps/w, 2090-gate NanoAES hardware accelerator with area-optimized encrypt/decrypt GF$(2^4)^2$ polynomials in 22nm tri-gate cmos. In: 2014 Symposium on VLSI Circuits Digest of Technical Papers. pp. 1–2 (June 2014)
8. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of AES. In: Paterson, K.G. (ed.) Advances in Cryptology – EUROCRYPT 2011. pp. 69–88 (2011)
9. National Institute of Standards and Technology (NIST): FIPS PUB 197: Advanced Encryption Standard (AES) (Nov 2001)
10. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with S-Box optimization. In: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology. pp. 239–254. ASIACRYPT '01 (2001)
11. Wamser, M.S., Sigl, G.: Pushing the limits further: Sub-atomic AES. In: 2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC). pp. 1–6 (Oct 2017)