Six shades of AES

Fatih Balli[1] and Subhadeep Banik[1]

LASEC, École Polytechnique Fédérale de Lausanne, Switzerland
{fatih.balli,subhadeep.banik}@epfl.ch

Abstract. Recently there have been various attempts to construct light weight implementations of the AES-128 encryption and combined encryption/ decryption circuits [2,13]. However no known lightweight circuit exists for AES-192 and AES-256, the variants of AES that use longer keys. Investing in lightweight implementations of these ciphers is important as we enter the post quantum era in which security is, by a rule of the thumb, scaled down to the square-root of the size of the keyspace. In this paper, we propose a single circuit that is able to offer functionalities of both encryption and decryption for AES-128/192/256. Our circuit operates on an 8-bit datapath and occupies around 3672 GE of area in silicon. We outline the challenges that presented themselves while performing the combinatorial optimization of circuit area and the methods we used to solve them.

1 Introduction

In the past few years, lightweight cryptography has become a popular research discipline. A number of lightwieght block ciphers have been proposed over the years. Among them Clefia [17] and Present [6] are well-studied with respect to their security and implementation. Both ciphers have been standardized in ISO/IEC 29192 "Lightweight Cryptography". Very recently the Simon and Speck family of block ciphers [5] was proposed by the NSA with the goal of reducing hardware area. While the above ciphers have mostly targeted optimization of hardware area, there have been other block ciphers aimed at optimizing other lightweight design metrics. For instance, the block cipher Prince [7] was designed for low latency applications like memory encryption. Another example is Midori [4] which was designed to optimize energy consumption. However, AES still remains the de-facto encryption standard worldwide for a number of sectors like banking and e-commerce. It is a part of several internet protocols like HTTPS, FTPS, SFTP, WebDAVS, OFTP, and AS2.

There have been several lightweight implementations of AES proposed in literature. In [16], the authors propose a 32-bit serial architecture with optimized tower field implementation of the S-box and a combinatorial optimization of the Mix Columns circuit. The size of this implementation was around 5400 GE (gate equivalents, i.e. are occupied by an equivalent number of 2-input NAND gates). The "Grain of Sand" implementation [12] by Feldhofer et al. constructs an 8-bit serialized architecture with circuit size of around 3400 GE but a latency of over

1000 cycles for both encryption and decryption. The implementation by Moradi et al. [15] with size equal to 2400 GE and encryption latency of 226 cycles is one of the smallest known architectures for AES. In [14], the authors report an 8-bit serial implementation that takes 1947/2090 GE for the encryption/decryption circuits respectively. This implementation makes use of intermediate register files that can be synthesized in the ASIC flow using memory compilers.

Very recently two further serial architectures have been proposed for AES-128. The first, named Atomic AES [2], which was followed up by Atomic AES v2.0. [3] uses the basic architecture of [15] along with a few tweaks to achieve encryption and decryption functionalities in the same circuit. The circuit takes around 2060 GE of area. [13] takes the design one step further, but proposing the first bit serial architecture for AES in less than 1600 GE. However since the architecture advances data one bit every clock cycle, it is around 8 times slower than byte serial architectures.

1.1 Motivation and Challenges

One important thing to note is that the all papers [15,2,3,13] assume that the key and data are input to the circuit arranged in a **row-major** fashion, i.e. bytes of each individual rows are input to the circuit together. This is slightly odd as AES specifications explicitly recommend **column-major** ordering and hence implementing AES in the proper columnwise ordering of bytes is an important challenge.

Secondly, there have surprisingly been no attempts made to implement AES-192 and AES-256 in a lightweight fashion. These are the variants of AES that use longer keys. Investing in lightweight implementations of these ciphers is important as we enter the post quantum era in which security is, by a rule of the thumb, scaled down to the square-root of the size of the keyspace, due to Grover's algorithm. Lightweight implementation of AES-192 and AES-256 is of added importance as AES-256 is a core component of a number of candidates in the NIST Post-quantum project for standardization of a quantum-secure public key cryptosystem [1]. NIST targets three level of security in this standardization: Level 1/3/5 respectively equivalent to AES-128/192/256 bit security. Out of 17 second round post quantum KEM candidate constructions, 9 candidates employ AES in their constrcution: 8 of these use AES-256 in counter mode, making it a preferred choice for generating pseudorandomness. Few candidates also propose 3 designs for 3 different security levels, therefore using all AES-128/192/256 instances at the same time. For signatures schemes, 2 out of 9 schemes make use of AES-256.

However designing serial implementations of AES-192 and AES-256 is slightly challenging due to reasons as outlined follows. One of the main reasons it is comparatively easy to implement AES-128 in a serial fashion is that the round function and key update operations are synchronized, which is to say that after every round, the state and the current key are updated. Thus every round involves executing the same operations on the state (except the last round MixColumns) and key registers which can be iterated 10 times to get the encryption/decryption functionality. This is however **not** the case with either AES-192 or AES-256. Since AES-192 uses a 192 bit key but only a 128 bit state, it requires only 8 full key update operations, to produce sufficient key material for the 12 rounds recommended by the designers. In fact, in AES-192 state and key operations become synchronized after 3 round functions and 2 key update operations. AES-256 uses a 256 bit key and requires only 7 full key updates for 14 executions of the round function. The key update operation of this cipher is also slightly different as each key update requires S-box operation to be applied on 2 columns of the current key instead of 1 in both AES-128 and AES-192. This asymmetry in the round and key schedule operations make serial implementation of AES-192 and AES-256 slightly more difficult.

A final challenge is implementing the functionalities of encryption and decryption on the same circuit. Various modes of operations like CBC [11] and EℓMD [10], that use block ciphers as the underlying primitive, require access to both its encryption and decryption functionalities. Thus it is useful to have an implementation that achieves both functionalities of a block cipher with minimal overhead.

1.2 Contribution and Organization

In this paper we present an 8-bit serial architecture that performs all encryption and decryption operations of three instances AES-128, AES-192, AES-256. The circuit thus supports six functionalities. We remove the requirement that bytes be ordered in row-first fashion, and construct our circuit so that it can support inputs when they are arranged in a column first fashion. The circuit occupies area of around 3672 GE when synthesized with the standard cell library of the STM 90nm CMOS logic process.

The paper is organized in the following manner. Section 2 gives some background on AES and some necessary definitions required to read the paper. Section 3 describes the architecture and functioning of our circuit in detail and we describe explicitly how we overcome some of the challenges presented in Section 1.1. Section 4 tabulates all implementation results and compares synthesis results for various standard cell libraries.

2 Background and Preliminaries

2.1 Encryption/Decryption Overview

Let r denote the number of rounds in the encryption/decryption function, n denote the number of key expansion rounds, ℓ denote the byte size of the key for a given AES instance. Note that, $r,\ n,\ \ell = (10, 10, 16),\ (12, 8, 24),$ and $(14, 7, 32)$ for AES-128, AES-192, and AES-256 respectively.

The encryption algorithm consists of multiple calls to AddRoundKey$(\cdot,\cdot)$, SubBytes$(\cdot)$, ShiftRows$(\cdot)$ and MixColumns$(\cdot)$ layers where each input denoted with '$\cdot$', as well as each output, is a 4×4 byte matrix. AddRoundKey takes the state information St and the round key K_i and returns the byte-wise XOR of them. With $\mathsf{SubBytes}$, each byte is substituted according to AES S-box. $\mathsf{ShiftRows}$ rotates the i-th row by i position to the left (for $i = 0, 1, 2, 3$). During $\mathsf{MixColumns}$, each column $[s_{4i}, s_{4i+1}, s_{4i+2}, s_{4i+3}]^T$ of input sequence s is multiplied with a fixed 4×4 byte matrix M, where byte values are treated as elements of $\mathsf{GF}(2^8)$. An important property of M is $M^4 = I$, where I denotes the identity matrix. We skip further details of these four layers, and refer the reader to [9].

Even though the block size in all 6 instances of AES is exactly 4×4 bytes and thereby matches exactly with the input and output of the aforementioned layers, the same cannot be said for the key. Namely, in $\mathsf{AES\text{-}192}$, the key is arranged in a 4×6 byte matrix and in $\mathsf{AES\text{-}256}$ it is arranged in a 4×8 byte matrix. As a consequence, the iterations of the key expansion algorithm, whose task is to generate fresh 4×4 bytes of round key for each $\mathsf{AddRoundKey}$ operation, desynchronize with the round operations performed on the state and it leads to a great deal of complexity in our design. We briefly remind the details of the key expansion algorithm below.

2.2 Key Expansion

Let S denote the AES S-box. Let $\mathsf{RC}_1, \ldots, \mathsf{RC}_{10}$ denote a sequence of round constant bytes[1]. The key expansion generates a sequence of key bytes $k_0, \ldots, k_{16r+15}$ given the bytes $k_0, \ldots, k_{\ell-1}$ as input. At each iteration of key expansion, ℓ bytes of *fresh* key is produced by XORing the original matrix with an additional offset. For instance, with $\mathsf{AES\text{-}128}$, the very first round of key expansion generates key bytes $k_{16}, \ldots, k_{31}$ from $k_0, \ldots, k_{15}$ according to:

$$\begin{bmatrix} k_{16}, k_{20}, k_{24}, k_{28} \\ k_{17}, k_{21}, k_{25}, k_{29} \\ k_{18}, k_{22}, k_{26}, k_{30} \\ k_{19}, k_{23}, k_{27}, k_{31} \end{bmatrix} \leftarrow \begin{bmatrix} k_0, k_4, k_8, k_{12} \\ k_1, k_5, k_9, k_{13} \\ k_2, k_6, k_{10}, k_{14} \\ k_3, k_7, k_{11}, k_{15} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S}(k_{13}) \oplus \mathsf{RC}_1, & k_{16}, k_{20}, k_{24} \\ \mathsf{S}(k_{14}), & k_{17}, k_{21}, k_{25} \\ \mathsf{S}(k_{15}), & k_{18}, k_{22}, k_{26} \\ \mathsf{S}(k_{12}), & k_{19}, k_{23}, k_{27} \end{bmatrix}$$

Similarly, with $\mathsf{AES\text{-}192}$, the very first round of key expansion is:

$$\begin{bmatrix} k_{24}, k_{28}, k_{32}, k_{36}, k_{40}, k_{44} \\ k_{25}, k_{29}, k_{33}, k_{37}, k_{41}, k_{45} \\ k_{26}, k_{30}, k_{34}, k_{38}, k_{42}, k_{46} \\ k_{27}, k_{31}, k_{35}, k_{39}, k_{43}, k_{47} \end{bmatrix} \leftarrow \begin{bmatrix} k_0, k_4, k_8, k_{12}, k_{16}, k_{20} \\ k_1, k_5, k_9, k_{13}, k_{17}, k_{21} \\ k_2, k_6, k_{10}, k_{14}, k_{18}, k_{22} \\ k_3, k_7, k_{11}, k_{15}, k_{19}, k_{23} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S}(k_{21}) \oplus \mathsf{RC}_1, & k_{24}, k_{28}, k_{32}, k_{36}, k_{40} \\ \mathsf{S}(k_{22}), & k_{25}, k_{29}, k_{33}, k_{37}, k_{41} \\ \mathsf{S}(k_{23}), & k_{26}, k_{30}, k_{34}, k_{38}, k_{42} \\ \mathsf{S}(k_{20}), & k_{27}, k_{31}, k_{35}, k_{39}, k_{43} \end{bmatrix}$$

However, $\mathsf{AES\text{-}256}$ contains a slight tweak (denoted with blue):

$$\begin{bmatrix} k_{32}, k_{36}, k_{40}, k_{44}, k_{48}, k_{52}, k_{56}, k_{60} \\ k_{33}, k_{37}, k_{41}, k_{45}, k_{49}, k_{53}, k_{57}, k_{61} \\ k_{34}, k_{38}, k_{42}, k_{46}, k_{50}, k_{54}, k_{58}, k_{62} \\ k_{35}, k_{39}, k_{43}, k_{47}, k_{51}, k_{55}, k_{59}, k_{63} \end{bmatrix} \leftarrow \begin{bmatrix} k_0, \ldots, k_{28} \\ k_1, \ldots, k_{29} \\ k_2, \ldots, k_{30} \\ k_3, \ldots, k_{31} \end{bmatrix} \oplus \begin{bmatrix} \mathsf{S}(k_{29}) \oplus \mathsf{RC}_1, & k_{32}, k_{36}, k_{40}, \mathsf{S}(k_{44}), k_{48}, k_{52}, k_{56} \\ \mathsf{S}(k_{30}), & k_{33}, k_{37}, k_{41}, \mathsf{S}(k_{45}), k_{49}, k_{53}, k_{57} \\ \mathsf{S}(k_{31}), & k_{34}, k_{38}, k_{42}, \mathsf{S}(k_{46}), k_{50}, k_{54}, k_{58} \\ \mathsf{S}(k_{28}), & k_{35}, k_{39}, k_{43}, \mathsf{S}(k_{47}), k_{51}, k_{55}, k_{59} \end{bmatrix}$$

The same operation is repeated for 10, 8, 7 times for $\mathsf{AES\text{-}128/192/256}$ respectively. Later, regardless of the instance and the initial key size, the subsequence $k_{16i}, \ldots, k_{16i+15}$ will act as the round key K_i for the i-th round. A key expansion round can be seen as a proper combination of the following unit operations, each of which processes one byte per clock cycle.

[1]Since at most 10 elements of this sequence is used, we consider it as a lookup table.

- ke0 (key expand 0) takes the byte from the second row and the last column of the current key, applies S-box, and XORs with the round constant RC_j. The result is added to the key byte in the first row and column e.g. $k_{16} \leftarrow k_0 \oplus \mathsf{S}(k_{13}) \oplus \mathsf{RC}_1$ is computed in AES-128. Each key expansion round contains exactly one ke0 operation.
- ke1 (key expand 1) takes the byte (from the next row) from last column and applies S-box, and XORs with the next key byte in the first column, e.g. $k_{17} \leftarrow k_1 \oplus \mathsf{S}(k_{14})$ is computed in AES-128. Each key expansion round contains exactly three ke1 operations.
- ke2 (key expand 2) takes the byte from the last column and the same row, applies S-box, and XORs with the original value, e.g. $k_{48} \leftarrow k_{16} \oplus \mathsf{S}(k_{44})$ is computed in AES-256. This operations is specific to AES-256 and is used exactly four times for each round of key expansion.
- kxor (key xor) XORs the current key byte with the $(\ell-4)$th previous keybyte, e.g. $k_{20} \leftarrow k_4 \oplus k_{16}$. Each key expansion round contains 12, 20, 24 kxor operations in AES-128, AES-192, AES-256 respectively.

The combination ke0, ke1, k1, ke1 performed for 4 consecutive clock cycles helps complete the first stage of the key expansion in which the last column of the current key is rotated, passed through the AES S-box, added to a round constant and thereafter added to the 1st column of the key. For AES-128 a keyschedule round consists of the following sequence of operations ke0, $\mathsf{ke1}^3$, kxor^{12}, where op^i denotes i successive executions of the operation op. For AES-192 the sequence is ke0, $\mathsf{ke1}^3$, kxor^{20} and for AES-256 the sequence is ke0, $\mathsf{ke1}^3$, kxor^{12}, $\mathsf{ke2}^4$, kxor^{12}. As already mentioned, the key expansion round and the encryption/decryption rounds are perfectly synchronized in AES-128, however the same cannot be said for AES-192 and AES-256. This was one of the primary challenges we had to overcome when designing a circuit that can perform all six instances together.

3 One Circuit to Rule Them All

3.1 Input, Output Formats

Our AES architecture is a sequential (clocked) one with 8-bit datapath. 8-bit KeyIn for key, 8-bit DataIn for the plaintext (resp. ciphertext) data, 3-bit selector Ins to choose among six instances (AES-128/192/256 encryption/decryption), a reset signal Rst and a clock signal Clk are wired as input. Its output consists of a 8-bit DataOut for the result (for the computed ciphertext or plaintext) and a ready signal Rdy indicating the completion of the operation. Loading the input values takes upto 16, 24, 32 cycles for AES-128/192/256 respectively, and the reception of the output takes 16 cycles; whereas the encryption/decryption operation takes on the order of few hundreds cycles.

We denote the data (i.e. the input plaintext/ciphertext) as a byte sequence $d_0, \ldots, d_{15}$. We denote the original key with $k_0, \ldots, k_{\ell-1}$ where ℓ is 16, 24, 32 for AES-128/192/256 respectively. Lastly, we denote the last ℓ bytes of round keys used in AddRoundKey with $k'_0, \ldots, k'_{\ell-1}$. Namely, these are the byte sequences

$k_{160}, \ldots, k_{175}$ in AES-128; $k_{184}, \ldots, k_{207}$ in AES-192; and $k_{208}, \ldots, k_{239}$ in AES-256.

Loading Cycles. In AES-128, the key and the data has the same size, therefore loading the both can be synchronized, i.e. k_i (resp. k'_i) and d_i are loaded at the same clock cycle for encryption (resp. decryption). However, in AES-192/256, the key is larger than the data, therefore we should clarify which bytes of the key and the data are loaded at which cycles. For encryption, the data and the first 16 bytes of key are loaded during the first 16 cycles. The remaining bytes of the key, i.e. $k_{16}, \ldots, k_{\ell-1}$, are loaded in the following 8 (resp. 16) cycles in AES-192 (resp. AES-256). For decryption, first $\ell - 16$ bytes of the last used round key bytes (the sequence $k'_0, \ldots, k'_{\ell-17}$) are loaded. Namely, first 8 (resp. 16) cycles are used to load $k'_0, \ldots, k'_7$ (resp. $k'_0, \ldots, k'_{15}$) in AES-192 (resp. AES-256). Then, the following 16 cycles are used to load $k'_{\ell-16}, \ldots k'_{\ell-1}$ and $d_0, \ldots, d_{15}$ simultaneously.

Input Format. For encryption, the key $k_0, \ldots, k_{\ell-1}$ and the data $d_0, \ldots, d_{15}$ are loaded. For decryption, the key byte sequence (see Section 2.2) $k'_0, \ldots, k'_{\ell-1}$ is loaded instead of the original key $k_0, \ldots, k_{\ell-1}$.

Result Cycles. The result data sequence $c_0, \ldots, c_{15}$ (ciphertext for encryption or plaintext for decryption) is observed at DataOut in the correct order. The signal Rdy is also set to '1' during the 16 cycles this result is available.

3.2 Components

- *Enabled byte flip flop* (henceforth referred as EFF) is a byte storage unit that preserves its output during many cycles when enable signal is unset, i.e. its value is frozen. When enable signal is set, its value (and thereby output) is updated with the first rising edge of the clock signal. They are denoted with shadowed white squares in Figure 1, and used in the key pipeline.
- *Enabled byte scan flip flop* (henceforth referred as SEFF) is an EFF combined with a multiplexer. Two separate bytes are wired as input, and its next value is assigned to either one of them based on an additional selection signal. Its value is updated on the next rising edge, if the enable signal is set. If enable signal is unset, its value is preserved. They are denoted with grayed and shadowed squares in Figure 1, and used mostly in the state pipeline.
- *Control Logic* is a finite-state machine which activates with the release of the reset signal Rst, and computes either one of the six AES instances based on Ins signal. It controls all flip flop enable signals, scan flip flop selectors, mux selectors, mask AND selectors, S-box direction signal and Rdy.
- *Mix Column* takes 4-byte column $[s_{4i}, s_{4i+1}, s_{4i+2}, s_{4i+3}]^T$ as input and computes $M \times [s_{4i}, s_{4i+1}, s_{4i+2}, s_{4i+3}]^T$ over $\mathsf{GF}(2^8)$, where M denotes the AES mixcolumn matrix [9]. Since $M^4 = I$ where I is the identity matrix, we can use the same circuit to do InvMixColumns by performing MixColumns three times. It outputs the 4-byte result.
- We use the Canright S-box architecture [8] that performs both the *S-box/S-box-inverse* operation and has a very low hardware footprint. S-box S, as

well as its inverse S^{-1}. The direction of the operation is determined with an additional selection signal.
- *RC lookup table* contains ten round constant bytes used in all three instances. A 4-bit counter is also attached to choose the correct value from the table.

In order to minimize number of gates, we limit our design to a single two-directional S-box (shared between SubBytes and KeyExpand), a single mix column circuit (used for both MixColumns and InvMixColumns) and a series of EFF and SEFF as two pipelines: one for the state and another for the key. Since the keysize in AES-256 is 32-bytes, the key pipeline contains 32 byte flip flops.

3.3 High Level Description of the Design

Our design is fully described in Figure 1. Below, we refer to EFF/SEFF directly though their two-digit addresses: for EFF/SEFF in the key pipeline we use 00, and for SEFF in the state pipeline we employ the italic font *00*.

State Pipeline. 16 SEFF are arranged in a upward-moving serial fashion, where a byte value enters into the pipeline from *33*, moves in the upwards direction to *30*, then moves to *23*, etc. and finally reaches to *00* in 16 clock cycles during normal operation. This is done via vertical connections in the pipeline which permit loading the state information in one-byte-per-clock fashion into the bus for executing AddRoundKey and SubBytes operations simultaneously in 16 cycles. Moreover, alternative lateral connections (e.g. from *30* to *00*) allow each column to be loaded into Mix Column circuit for MixColumns operation in 4 cycles. The same lateral connections in the left direction allows us to do ShiftRows operation, by carefully enabling and disabling rows in harmony in 3 cycles. With the help of muxes connected to *30, 31, 32, 33*, we can choose between ShiftRows and MixColumns operations. Notice that the control logic determines the direction of the flow by the select signals and whether or not some EFF/SEFFs are frozen by enable signals. Partially or fully freezing is useful for ShiftRows or when another operation is stalling the key pipeline.

Key Pipeline. It consists of 31 EFF and 1 SEFF to store the 32 byte key in AES-256. The connections of the pipeline are tweaked through muxes 5, 6 in such a way that:

- During AES-128 operations, from 20 to 53 are bypassed (disabled) and the output of 60 is wired to 13 through mux 5. Therefore the key pipeline effectively shrinks to 16 byte flip flops.
- During AES-192 operations, from 40 to 53 are bypassed (disabled) and the output of 60 is wired to 33 through mux 6. The output of 20 is wired to 13 through mux 5. The key pipeline shrinks to 24 flip flops.
- During AES-256 operations, no EFF in the key pipeline is disabled and 40 is wired to 33 through mux 6 and 20 is wired to 13 through mux 5.

In order to work in harmony with the state pipeline, the task of the key pipeline is to provide the particular byte of key to the bus, so that AddRoundKey can be performed correctly with the byte coming from the state. This key byte from the pipeline can be fetched from 00, 20 or 40 based on the selection signal of mux 10, whereas the pipeline supports rotation through connections 00 $\rightarrow$ 73 through mux 12. As before, enable signals are configured by the control logic and can freeze the pipeline when another operation is stalling the state pipeline.

Main Bus. Consists of two muxes 10, 7 to choose the source of key and data bytes to be loaded into the bus. The crucial component in the bus is the S-box, whose input and output is complemented with two byte XOR gates. The XOR gate before S-box is useful for encryption, as the key addition precedes the S-box, and the XOR gate after S-box is useful for decryption. The choice of S-box/S-box-inverse functionality and the select signals of muxes are configured by the control logic.

Key Expansion Logic. The most challenging part of our design by far is the computation of proper round key for AddRoundKey operation for 6 different instances on the same circuit. For this reason, a combination of XOR/AND gates is connected to the key pipeline to execute KeyExpand on-the fly (while the pipeline is moving to perform another operation). The gates highlighted with lightgray background in Figure 1 connected to 00, 10 (positioned above key pipeline) enables key expansion for the encryption and decryption, and the gates connected to 13, 23, 33, 63, 73 (positioned below the key pipeline) enable key expansion during decryption.

3.4 Elementary Operations of Layers

In order to simplify the explanation of how our circuit operates, we conceptually divide the control of the circuit into various operations. We also explain their connection to four different layers (plus KeyExpand). Some of the operations described below are computed on completely independent parts of the circuit, hence they can be performed simultaneously by our hardware. We will squeeze them into same cycles as much as possible. Each of the following instructions sets particular control bits for given cycle to perform its corresponding operation. If an operation does not explicitly mandate how a certain SEFF/EFF should behave, then it is **frozen** by setting the enable signal to '0'. As before, 00 refers to the top-left EFF of the key pipeline and *00* refers to the top-left SEFF in the state pipeline.

add Both the key and the state pipelines are fully active, and two bytes from each are loaded into the bus. The state byte is fetched from *00* of the state pipeline. On the other hand, the key byte can be fetched either from 00, 20 or 60 of the key pipeline (note that key bytes are fetched from 20 and 60 during AES-192 decryption). Exception to this is the initialization where the key and the data are being loaded to the circuit: then, two bytes must come from DataIn and KeyIn but not from the pipelines. If the chosen functionality of the circuit indicated by Ins signal is encryption, the two bytes on the bus are first XORed, and then passed through S-box (therefore AddRoundKey and

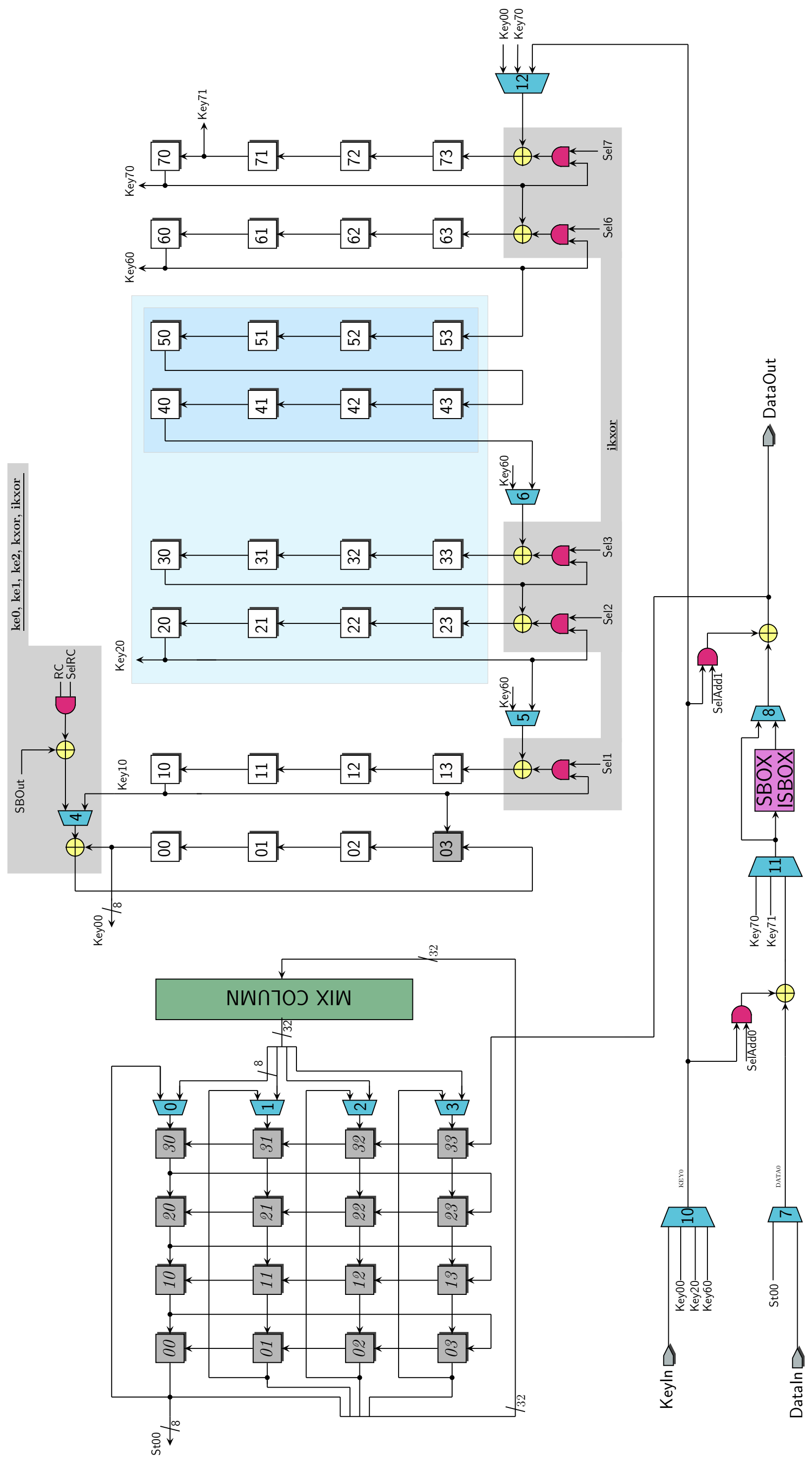

Fig. 1: Circuit diagram for the 6AES architecture

SubBytes are done concurrently). Otherwise (if Ins indicates decryption), the state byte is passed through S-box-inverse, and then the addition is done (therefore InvSubBytes and AddRoundKey are done concurrently). In either case, the computed byte is stored to *33* of the state pipeline. Meanwhile, the key pipeline rotates itself by connecting 00 to 73. Exception to this is again initialization, during which 73 receives its next value from the bus.

sbox Muxes $11, 8$ and S-box selection signal are configured accordingly so that S-box can be computed.

isbox Muxes $11, 8$ and S-box selection signal are configured accordingly so that S-box-inverse can be computed. Both sbox and isbox are performed simultaneously with add during encryption/decryption operations respectively.

srow0 Rotates rows *1*, *2*, *3* of the state pipeline to left by one. The control logic uses selection signal of scan flip flops to change the direction in the pipeline, and freezes the unused state flip flops.

srow1 As before, but rotates rows *1*, *2* of the state pipeline to left by one.

srow2 As before, but rotates rows *3* of the state pipeline to left by one. Notice that consecutive srow0, srow1, srow2 operations (3 cycles) correspond to one ShiftRows.

isrow0 As before, but rotates row *1* of the state pipeline to left by one.

isrow1 As before, but rotates rows *1*, *2* of the state pipeline to left by one. Notice that consecutive isrow0, isrow1, srow0 operations (3 cycles) correspond to one InvShiftRows.

mixcol Muxes $0, 1, 2, 3$ are configured to load the input from Mix Column circuit. Again, the selection signal of all state flip flops are configured by the control logic so that the pipeline moves in the left direction.

ke0 Performs the key expand operation as explained in Section 2.2. During ke0, all flip flops of the key pipeline except columns 0 and 7 are frozen. Columns 0 and 7 rotate in the upwards direction. The state pipeline is also frozen.

ke1 The only difference from ke0 is that SelRC is set to 0, so that RC is removed from the computation, 03 is loaded with $\mathsf{S}(71) \oplus 00$.

ke2 Similar to ke0, but the input byte of S-box is not rotated, 03 is loaded with $\mathsf{S}(70) \oplus 00$.

kxor For key xor operation of the key expansion algorithm, the input select bits of 03 and the mux 4 are configured to store $10 \oplus 00$ instead of barely 10 for the next cycle.

ikxor For the inverse of key xor operation used in decryption we use the same trick employed in [2] in which the last row of byte flip-flops in the key register is controlled with additional and gates. The corresponding circuitry is shown in a gray background in Figure 1. Sel1, Sel2, Sel3, Sel6, Sel7 are the corresponding signals that are configured such that key XOR is done, e.g. $13 \leftarrow 10 \oplus 20$, only at selected clock cycles during decryption. Similar to kxor, the key pipeline must be fully active, and state pipeline is frozen.

load Mux 10 is configured such that the key is loaded from the input to the pipeline (necessary for AES-192, AES-256). The key pipeline is fully active, and the state pipeline is frozen.

rot The key is rotated in the pipeline, where the exiting byte 00 is fed back into 73. The key pipeline is fully active.

rxor Pseudonym for combination of rot and kxor. Therefore the key is updated on the pipeline with key xor operation, as it rotates.

In the following two subsections, we will separate encryption/decryption round functions performed on the state, completely from key expansion. Encryption and decryption round function operations are straightforward to implement with the design given in Figure 1 and remains quite similar through six different instances. However, the key expansion becomes a major challenge and due to its instance-specific nature, requires significant effort.

3.5 Generic Encryption/Decryption Overview

First, for the sake of argument, suppose that the key pipeline always contains the necessary round key K_i at round i, with which AddRoundKey is being done. Then we can readily convert the encryption algorithm into a sequence of operations. AddRoundKey and SubBytes can be done simultaneously through add and sbox operations in 16 cycles. Then for ShiftRows, it suffices to run srow0, srow1, srow2 subsequently in 3 cycles. Then, in 4 cycles of mixcol, we complete MixColumns. This sequence corresponds to one round of operation in the encryption algorithm, and can be repeated as many times as necessary, as long as the key pipeline handles the key expansion and provides the correctly aligned key bytes during AddRoundKey. The same line of reasoning also applies to decryption, where InvSubBytes and AddRoundKey can be done with isbox and add simultaneously in 16 cycles, InvShiftRows can be done with isrow0, isrow1, srow0 in 3 cycles; and InvMixColumns can be done in 12 cycles of mixcol (as explained before InvMixColumns is 3 repetitions of MixColumns).

Therefore, what remains is to continuously *refresh* the key in the pipeline, by removing *dirty* key bytes (i.e. already used with AddRoundKey), and replacing with *fresh* bytes (not yet used) of key. By refreshing we mean computing the next round key in encryption, and previous round key in decryption (since decryption starts with the last ℓ key bytes of the last round and computes the round keys in the reverse direction). In the following section we describe how key bytes are managed in the key pipeline, and how its operations are interleaved with the four layers of encryption and decryption.

3.6 Key Expansion Details

AES-128 Encryption: The detailed chronology of operations is given in Figure 2. During the first 16 cycles, muxes $7, 10$ are configured such that the key and the data are loaded to the bus through inputs DataIn, KeyIn, instead of the pipelines. At the same time, AddRoundKey and SubBytes operations are done simultaneously, where the computed state is loaded into the state pipeline, and the key is loaded into 00-13, 60-73.

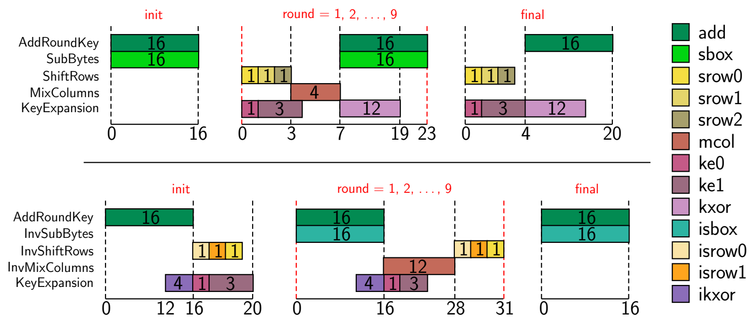

Fig. 2: The chronology of operations in AES-128 encryption (on top) and AES-128 decryption (below). The numbers in the boxes indicate the number of cycles over which the operation is executed.

A round takes 23 cycles to complete. At the beginning of the round, all the keys in the pipeline are dirty. Therefore, we use the first 4 cycles to refresh the key bytes in column 0 with running ke0, ke1, ke1, ke1 sequentially. ShiftRows and MixColumns are also performed in parallel, since they have no effect on the key pipeline. At the end of 7 cycles (after ke0, ke1^3), the key pipeline still contains 12 dirty key bytes contained in 10-13, 60-73. These bytes are refreshed in 12 cycles with kxor as the pipeline moves, as they are loaded into 03. Therefore, it is merged with add and sbox, which takes 16 cycles. At the end of a round, all bytes in the key pipeline are again dirty. In the final round, MixColumns is skipped, and the ciphertext is available during the very last 16 cycles.

AES-128 Decryption: We remind that for decryption, KeyIn loads the very last 16 bytes of key used with the last AddRoundKey, but not the original key used for encryption. The rounds can be seen as the symmetrically opposite versions of encryption.

A round takes 31 cycles to complete. At the beginning, all bytes in the key pipeline are fresh. At the end of 12 cycles, the key pipeline contains only 4 fresh bytes. Then, ikxor is enabled through 13, 63, 73 (by setting Sel13, Sel63, Sel73 to '1') for 4 cycles. Therefore at cycle 16, the key pipeline contains exactly 12 bytes of fresh key contained in 10-13, 60-73. The remaining dirty key column is refreshed by ke0, ke1, ke1, ke1 operations are in the next 4 cycles. Therefore, at the end of the round, all bytes in the key pipeline are fresh. As before, the output of decryption, i.e. the plaintext becomes available during the last 16 cycles.

AES-192 Encryption: The detailed chronology of operations is given in Figure 3. Performing the key expansion in AES-192 becomes quite challenging given the fact that each key expansion round generates 24 bytes of new round key, whereas only 16 of them are used for each encryption round. This leads to misalignment and desynchronization issues between the state pipeline and the key pipeline.

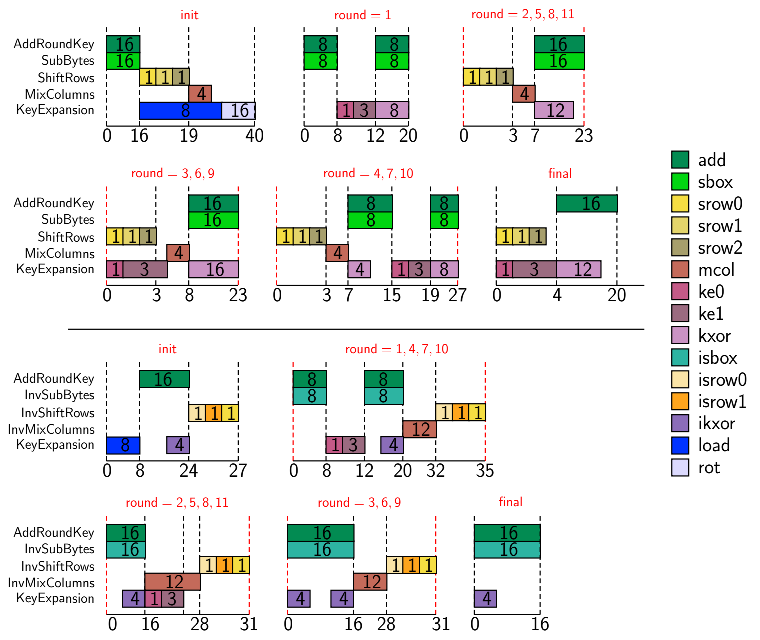
Fig. 3: The chronology of operations in AES-192 encryption (on top) and AES-192 decryption (below). The numbers in the boxes indicate the number of cycles over which the operation is executed.

We overcome them by interrupting AddRoundKey and SubBytes operations and running key expansion algorithm in the middle. This leads to three different types of rounds: (1) first type of round has no fresh key byte in the pipeline at the beginning and has to run a key expansion round algorithm before addition, (2) the second type of round has 4 leftover fresh bytes in 00-03 and 4 dirty bytes in 10-13 that can be refreshed with kxor as the pipeline moves. This means that AddRoundKey and SubBytes have to run for 8 cycles, then pause for key expansion, and later resume for 8 more cycles (3) third type of round has 4 fresh bytes in 00-03, and 12 dirty bytes in 10-33 that can be refreshed with kxor as the pipeline moves.

During the first 16 cycles, AddRoundKey and SubBytes is simultaneously performed as before. The next 8 cycles are used to load the rest of the key into the key pipeline. Then, in order to align the key properly, the key pipeline is rotated for 16 cycles with rot. Thereby, at the end, 8 fresh bytes are located at 00-13, and the dirty bytes are at 20-33, 60-73.

During round 1, we have to interrupt AddRoundKey and SubBytes after 8 cycles, at which point all the bytes in the key pipeline are dirty. The 4 bytes of key that requires to be updated by key expand 0 and key expand 1 operations are located at 00-03, therefore we run ke0, ke1, ke1, ke1 in the following four cycles. The remaining 20 dirty key bytes are refreshed as they are loaded into

03, by running kxor alongside add and sbox operations, and it overflows into the next round. Of this 8 are done in the current encryption round and 12 are deferred to the next round. Note that since 8 AddRoundKey operations are done simultaneously, at the end of this round the number of fresh bytes in the key pipeline is $4 + 8 - 8 = 4$.

At the beginning of rounds 2, 5, 8, 11 (which are type (3) rounds) the pipeline contains only 4 bytes of fresh key, but the following 12 dirty bytes can be refreshed with kxor. Therefore, to align correctly, one should run kxor during the first 12 cycles of AddRoundKey and SubBytes. At the end of this round, all fresh bytes are therefore used up.

At the beginning of rounds 3, 6, 9 (which are type (1) rounds); the key in the pipeline is completely dirty and the first column requires key expansion 0 and key expansion 1 operations. Therefore ke0, ke1, ke1, ke1 are run in the first 4 cycles. The following 20 bytes of key can be easily refreshed with kxor alongside add and sbox. Of this 16 is executed in the current round and 4 are deferred to the next round.

At the beginning of rounds 4, 7, 10 (which are type (2) rounds) there are 4 bytes of fresh keys followed by 4 bytes of dirty keys that can be refreshed with kxor in the key pipeline. However, the following column of key requires the key expand 0 and key expand 1 operations, so add and sbox is interrupted as before for key expansion. The remaining 8 bytes of addition continues after 4 cycles of ke0, ke1, ke1, ke1. The ciphertext is available in DataOut during the last 16 cycles.

AES-192 Decryption: A second obstacle that arises during the decryption is that fresh bytes in the key pipeline are not necessarily always start from 00. Recall that for decryption, the last 24 bytes of used round keys are loaded initially, therefore we have to run the key expansion algorithm in the reverse order. Therefore, we have to start refreshing key columns starting with the highest index, i.e. whichever column of key was used last in the encryption should be removed first. At the same time, due to flow direction of the pipeline, the lowest indexed key column occupies 00-03, whereas during various stages of operation the key columns to be used in key addition are located at 20-23 or 60-63. Our solution is to connect pipeline exits Key20, Key60 to mux 10, so that even if the next fresh key byte is misaligned in the key pipeline, we can continue AddRoundKey, InvSubBytes operations without requiring additional cycles for rotation. This irregular exit of key bytes from the pipeline is only necessary for AES-192 decryption.

Since the last 24 bytes of round key is loaded into the circuit (k_{184} to k_{207}), 8 cycles are used for loading the first 8 bytes of this key. Then the following 16 cycles are used for add. During the last four cycles of add, ikxor is also performed through 33, 63, 73 (but not 03, 13, 23). Therefore, at cycle 24, the key pipeline contains 20 fresh bytes (8 unused from the initial load and 12 from ikxor), where the 4 dirty bytes are stored in 20-23 and they can only be refreshed with ke0, ke1, ke1, ke1. Therefore, we will wait until this key moves into 00-03.

At the beginning of rounds 1, 4, 7, 10; the key pipeline contains 20 fresh bytes. However the next 8 fresh bytes to be used for add are located at 60-73, whereas the remaining 8 bytes required for add are located at 00-13. Therefore we fetch the next byte key into the bus from Key60, and at the same time rotate the pipeline by connecting 00 $\rightarrow$ 73. After 8 cycles, we interrupt isbox and add because the dirty column of key that requires the key expand 0/1 operations to update now reaches 00-03, so we can perform ke0, ke1, ke1, ke1. After refreshing this column of keys in 4 cycles, we resume fetching key bytes from 60 for AddRoundKey and InvSubBytes. Concurrently, at the last 4 cycles, we do ikxor with 03, 13, 23, 33 to obtain 16 fresh bytes for the next round. All the 24 bytes in the pipeline after this are completely fresh.

At the beginning of rounds 2, 5, 8, 11; the key pipeline is completely fresh. However the next 16 bytes of key to be used with add are located at 20-33 and 60-73. Therefore, key bytes are fetched from Key20 into the pipeline, and the pipeline is rotated as before. During the last 4 cycles of add, ikxor is performed over 13. After the 16 add cycles, the bytes of key that require update by key expand 0/1 arrive at 00-03, and therefore ke0, ke1, ke1, ke1 is executed to generate 4 fresh bytes. At the end, the key pipeline contains 16 fresh key bytes in 00-33.

At the beginning of rounds 3, 6, 9; the key pipeline contains 16 fresh bytes starting from 00, and they are aligned with the state pipeline for add. In order to arrange future key bytes, we still perform ikxor on 33, 63 for the first 4 cycles, and 33, 63, 73 for the last 4 cycles. At the end, the key pipeline contains 4 dirty bytes located at 20-23.

AES-256 Encryption: The detailed chronology of operations is given in Figure 4. AES-256 remains simpler to achieve than AES-192, because each key expansion round produces enough keys for two AddRoundKey operations. During the first 16 cycles, add, sbox is performed. We spend other 16 cycles to load the rest of the key. Then the key expansion is performed with ke0, ke1, ke1, ke1, and key is rotated for 16 cycles to move the fresh key bytes to 00-33. During the first 12 cycles of this period, we also enable kxor (named rxor for convenience) so that old keys are refreshed as they rotate through the pipeline.

At the beginning of round 1, the key pipeline is completely fresh, therefore there are sufficient bytes of keys for round 2 as well. Therefore, no key expansion operation is done during the first two rounds.

At the beginning of rounds 3, 5, 7, 9, 11, 13; the key pipeline is completely dirty, and the bytes at 00-03 require 4 cycles of ke2. Then at the first 12 cycles of add, kxor is also enabled so that the following 12 dirty bytes can be refreshed.

The rounds 4, 6, 8, 10,12 work exactly same, except the special key column requires ke0, ke1, ke1, ke1 rather than 4 cycles of ke2. The ciphertext is available in the last 16 rounds of the final round.

AES-256 Decryption: Since the last 32 used bytes of key are loaded into the circuit, we use first 16 cycles to load the first half of this key. The next 16 cycles receives the data and the second half of the key at the same time, therefore

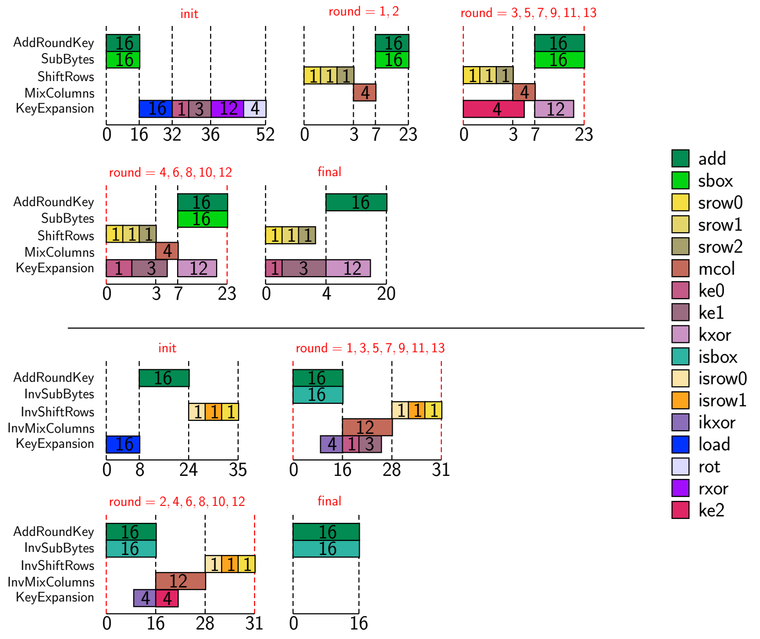

Fig. 4: The cycle arrangement of AES-256 encryption (on top) and AES-128 decryption (below). The numbers in the boxes indicate the number of cycles over which the operation is executed.

performs the add operation. At the beginning of rounds 1, 3, 5, 7, 9, 11, 13; the first 16 bytes of the key pipeline are fresh and the rest is dirty. At the last 4 cycles of add, ikxor is performed through 13, 23, 33 so that 12 bytes are refreshed. The following 4 bytes are also refreshed with ke0, ke1, ke1, ke1. The rounds 2, 4, 6, 8, 10, 12 work exactly same except the key column requiring update by key expand 2 is refreshed with 4 cycles of ke2 instead of ke0, ke1, ke1, ke1. The plaintext is available in the last 16 rounds of the final round.

4 Performance Evaluation and Conclusion

In order to perform a fair performance evaluation, we implemented the circuit using VHDL. Thereafter the following design flow was adhered to for all the circuits: a functional verification at the RTL level was first done using Mentor Graphics Modelsim software. The designs were synthesized using the standard cell libraries of the CMOS logic processes listed in Table 1, with the Synopsys Design Compiler, with the compiler being specifically instructed to optimize the circuit for area. A timing simulation was done on the synthesized netlist to confirm the correctness of the design, by comparing the output of the timing simulation with known test vectors. The switching activity of each gate of the circuit was collected while running post-synthesis simulation. The average

Table 1: Performance Comparison of 6AES architecture (E: Encryption, D: Decryption). Power is reported at a clock frequency of 10 MHz. $\mathsf{TP}_{\mathsf{max}}$ denotes the maximum throughput achievable on the circuit.

Library	Area	Power	Variant	Latency	Energy	$\mathsf{TP}_{\mathsf{max}}$	Variant	Latency	Energy	$\mathsf{TP}_{\mathsf{max}}$
		(GE)	(μW)		(cycles)	(nJ)	(Mbps)	(cycles)	(nJ)	(Mbps)
STM 90nm	3672	189.5	AES-128E	243	4.605	75.8	AES-128D	315	5.969	58.5
			AES-192E	322	6.102	57.2	AES-192D	400	7.580	46.0
			AES-256E	371	7.030	49.6	AES-256D	454	8.603	40.6
TSMC 90nm	4760	95.4	AES-128E	243	2.318	71.8	AES-128D	315	3.005	55.4
			AES-192E	322	3.072	54.2	AES-192D	400	3.816	43.7
			AES-256E	371	3.539	47.1	AES-256D	454	4.331	38.5
UMC 90nm	5009	192.9	AES-128E	243	4.687	101.3	AES-128D	315	6.076	78.1
			AES-192E	322	6.211	76.4	AES-192D	400	7.716	61.5
			AES-256E	371	7.157	66.3	AES-256D	454	8.758	54.2
TSMC 180nm	4680	1209.9	AES-128E	243	29.400	71.5	AES-128D	315	38.112	55.1
			AES-192E	322	38.959	53.9	AES-192D	400	48.396	43.4
			AES-256E	371	44.887	46.8	AES-256D	454	54.929	38.2

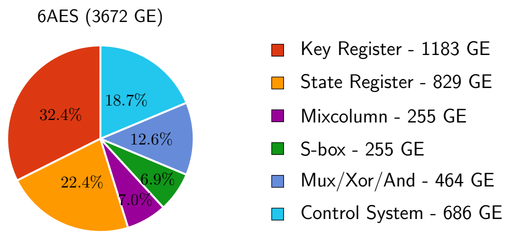

Fig. 5: Area requirements of the individual components

power was obtained using *Synopsys Power Compiler*, using the back annotated switching activity.

We outline some of the essential lightweight metrics of the 6AES architecture in Table 1. In Figure 5, we present a component-wise breakdown of the circuit size when synthesized with the STM 90nm logic process. A significant area is required for generating the control signals, as accommodating 6 different functionalities in a single circuit requires more fine-grained control over specific circuit components. This is because both the structure (wrt sequence of operations) and duration (wrt number of clock cycles) of a single round shows a wide range of variations as the size of the key changes. To the best of our knowledge, this is the first work that aims ot minimize the size of the circuit while implementing all the 3 versions of the AES circuit. The circuit offers flexibility to designers who want to move to higher levels of security in the near future, and implement modes of operation that would require simultaneous access to block cipher encryption/decryption circuits.

Acknowledgments: Subhadeep Banik is supported by the Ambizione Grant PZ00P2_179921, awarded by the Swiss National Science Foundation.

References

1. NIST Post-Quantum Cryptography Project. Available at `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`
2. S. Banik, A. Bogdanov, F. Regazzoni. Atomic-AES: A Compact Implementation of the AES Encryption/Decryption Core. In INDOCRYPT 2016, LNCS, vol. 10095, pp. 173-190, 2016.
3. S. Banik, A. Bogdanov, F. Regazzoni. Atomic-AES v 2.0. In IACR eprint archive. Available at `https://eprint.iacr.org/2016/1005.pdf`.
4. S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, F. Regazzoni. Midori: A Block Cipher for Low Energy. In ASIACRYPT 2015, LNCS, vol. 9453, pp. 411-436, 2015.
5. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, L. Wingers. The Simon and Speck Families of Lightweight Block Ciphers. In IACR eprint archive. Available at `https://eprint.iacr.org/2013/404.pdf`.
6. A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In CHES 2007, LNCS, vol. 4727, pp. 450-466, 2007.
7. J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knežević, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, T. Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In Asiacrypt 2012, LNCS, vol. 7658, pages 208-225, 2012.
8. D. Canright. A very compact S-Box for AES. In CHES 2005, LNCS, vol. 3659, pp. 441-455, 2005.
9. J. Daemen, V. Rijmen. The design of Rijndael: AES - the Advanced Encryption Standard. Springer-Verlag, 2002.
10. N. Datta and M. Nandi. ELmD v1.0. Submission to the Caesar compedition. Available at `https://competitions.cr.yp.to/round1/elmdv10.pdf`.
11. M. Dworkin. Recommendation for Block Cipher Modes of Operation. NIST Special Publication 800-38A. Available at `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.
12. M. Feldhofer, J. Wolkerstorfer, V. Rijmen. AES Implementation on a Grain of Sand. In IEEE Proceedings of Information Security, vol. 152(1), pages 13-20, 2005.
13. J. Jean, A. Moradi, T. Peyrin, P. Sasdrich. Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives - Applications to AES, PRESENT and SKINNY. In CHES 2017, LNCS, vol. 10529, pp. 687-707, 2017.
14. S. Mathew, S. Satpathy, V. Suresh, M. Anders, K. Himanshu, A. Amit, S. Hsu, G. Chen, R.K. Krishnamurthy. 340 mV–1.1V, 289 Gbps/W, 2090-gate nanoAES hardware accelerator with area-optimized encrypt/decrypt $GF(2^4)^2$ polynomials in 22 nm tri-gate CMOS. In IEEE Journal of Solid-State Circuits, vol. 50, pp. 1048–1058, 2015.
15. A. Moradi, A. Poschmann, S. Ling, C. Paar, H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Eurocrypt 2011, LNCS, vol. 6632, pp. 69-88, 2011.
16. A. Satoh, S. Morioka, K. Takano, S. Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In Asiacrypt 2001, LNCS, vol. 2248, pp. 239-254, 2001.
17. T. Shirai, K. Shibutani, T. Akishita, S. Moriai, T. Iwata. The 128-bit Block-cipher CLEFIA(Extended Abstract). In FSE 2007, LNCS, vol. 4593, pp. 181-195, 2007.