Universiteit Leiden
The Netherlands

Opleiding Informatica

Optimizing AES for RISC-V Cores

Imke van Ooijen

Supervisors:
Todor Stefanov & Abolfazl Sajadi

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/07/2024

Abstract

The PROACT project, a collaboration between Leiden University, Radboud University, and Riscure, aims to enhance physical attack resistance on IoT devices by developing new silicon chips and cryptographic algorithms, as well as optimizing existing algorithms. This thesis presents an optimized implementation of the Advanced Encryption Standard (AES) encryption and decryption algorithm for the PROACT chip, which features 32-bit RISC-V Ibex cores. The implementation in C completes encryption in 1218 clock cycles, which is only 3.4% slower than the existing assembly implementation while offering improved readability, portability, and extensibility. Furthermore, it extends the functionality by including decryption functionalities, as well as cipher block chaining (CBC) and counter (CTR) modes of operation. A 10% performance improvement was achieved in CTR mode through known optimization techniques from the literature. Additionally, a graphical user interface has been developed to facilitate testing and further research on the PROACT chip.

Contents

1 Introduction

Due to the rapid growth of the Internet of Things (IoT), an increasing amount of devices use the internet to communicate, which also increases the need for adequate protection mechanisms for these devices. Physical Attack Resistance of Cryptographic Algorithms and Circuits with Reduced Time to Market (PROACT) is a collaborative research project including Leiden University, Radboud University and Riscure that aims at developing new silicon chips and algorithms that enhance the protection against physical attacks on IoT devices. An example of a physical attack is a side-channel power analysis attack [RD20], where the power consumption of a device that is performing cryptographic operations is monitored. The collected data can be used to infer the private encryption key, which can lead to privacy violations. To identify the possibility and prevent attacks like these, researchers within the PROACT project have been developing a silicon chip that facilitates the collection of power traces in order to analyze side-channel power leakage. The prototype of this chip features two RISC-V 32-bit Ibex cores [Saj23]. One core serves as a controller to set up a wide range of experiments on the chip, while the other is dedicated to executing cryptographic algorithms that are part of the experiments. Each core is equipped with a separate 64 KB data and instruction memory.

This thesis focuses on developing an Advanced Encryption Standard (AES) [NIS01a] encryption and decryption software implementation optimized for the PROACT chip, featuring cipher block chaining and counter mode as modes of operation. The AES algorithm has been one of the most widely used encryption algorithms since it became the standard in 2001 [NIS01b]. Considering that the PROACT chip will mainly be concerned with running cryptographic algorithms, it is important that there is an AES implementation which efficiently utilizes the available resources on this chip.

The designers of the AES algorithm themselves described an efficient implementation on 32-bit platforms using look-up tables. While offering performance improvement, table-based implementations are generally vulnerable to cache-based timing attacks [Ber05, BM06]. However, the PROACT chip does not have a data cache and is thus assumed to be excluded from this vulnerability. Therefore, this work employs a table-based AES approach. An optimized implementation of AES encryption for the RISC-V instruction set architecture has been proposed in [Sto19]. However, this implementation was developed using RISC-V assembly for a key size of 128 bits and does not feature any modes of operation. While developing encryption algorithms using assembly code can be very efficient in terms of performance, such implementations lack the readability, maintainability, portability, and expandability that is offered by using higher-level languages like C. In addition to this advantage of high-level languages, modern compilers have advanced capabilities for optimizing code during the compilation stage. Therefore, this thesis explores the possibilities of developing AES encryption in C while maintaining a performance comparable to that of the assembly implementation. Efforts are made to write C code in a manner that allows the compiler to recognize certain optimizations, thereby generating more efficient assembly code. Based on the difference in the performance of the implementations, the decryption and modes of operation are implemented and optimized in either RISC-V assembly or C. The performance of the implementations is measured in clock cycles on an Ibex emulator and is verified on the PROACT chip prototype on an FPGA board.

1.1 Contributions

The two main contributions of this thesis are:

- A C-based implementation of AES encryption and decryption featuring cipher block chaining and counter mode, optimized for the PROACT chip. This implementation can also be used for other RISC-V-based platforms that do not feature a data cache.
- A graphical user interface that enables the user to experiment with cryptographic algorithms on the PROACT chip by providing their source code and the data to encrypt or decrypt. This user interface is beneficial for further research and development on the PROACT chip.

1.2 Thesis overview

Section 2 gives an overview of the research that has already been conducted in this field. Section 3 explains the functionality of the AES algorithm and provides the specifications of the RISC-V ISA and the PROACT chip. In Section 4, the methodology to develop and optimize the AES implementation is discussed. Section 5 describes and interprets the results of the experiments. Section 6 summarizes the key findings and limitations of this work and offers concluding remarks.

This bachelor thesis is supervised by Dr. T.P. Stefanov and Mr. A. Sajadi at the Leiden Institute of Advanced Computer Science. The source code for the developed AES implementations and the graphical user interface is available at `https://git.liacs.nl/s3293912/bachelor_thesis`.

2 Related Work

Since the Rijndeal block cipher [DR02] was announced as the AES, research efforts have been directed towards optimizing the algorithm in various contexts. This section offers an overview of these efforts while also highlighting the scarcity of studies specifically focused on RISC-V architecture.

Efficient AES implementations for ARM-based platforms are discussed in [OBSC10] and [ABM04]. While both the ARM and the RISC-V ISA employ a reduced instruction set, the former is more elaborate. For example, the ARM ISA provides instructions to shift or rotate data before it is processed without any penalty to the number of instruction cycles. The proposed implementations make heavy use of these instructions, that are not supported by the RISC-V ISA. Thus, these optimizations can not be directly translated to the target RISC-V ISA of this work.

A general optimization technique which can be applied to AES-CTR implementations is counter-mode caching, where values that are unchanged between multiple blocks are saved and reused. This technique was first introduced for the eSTREAM AES implementation [Wu07]. The authors of [PL18] developed this technique further and claim a performance improvement of 15%–20% for table-based AES. Their test environments are more advanced compared to the PROACT chip prototype. On the low-resource Ibex cores that are used in this work, the proposed techniques lead to a 10% improvement in performance.

Little research has been done into optimizing the AES algorithm specifically for the RISC-V ISA. In [BS08], several architecture-dependent techniques are discussed to optimize the table-based AES implementation by reducing the number of instructions. However, most of these techniques can not be used for the limited instruction set of the RISC-V architecture. In [Sto19], the proposed baseline implementation and the applicable optimization techniques are applied to develop the first AES encryption implementation optimized for the RISC-V architecture. However, as mentioned in the introduction, this implementation features neither decryption nor modes of operation. Furthermore, readability, maintainability, portability, and extensibility are sacrificed by using assembly code instead of a higher-level language. The optimized assembly implementation has been translated into C [CJL+20], but due to the literal translation approach that was taken to perform this translation, only a slight improvement is provided in the listed shortcomings of the assembly code. On the contrary, this thesis focuses on maintaining the benefits of high-level programming languages and extending the existing partial implementations into complete AES implementations.

Research on non-table-based AES is conducted in [AP20], setting new bit-sliced AES speed records for the RISC-V ISA. However, the bit-sliced implementation is still slower compared to the table-based approach. Since the PROACT chip does not have a cache, this work explores the possibilities of using a table-based approach to achieve greater efficiency.

3 Background

This section gives an overview of the relevant background for this thesis. Its primary focus lies on the AES algorithm, discussing the functionality of both the encryption and decryption ciphers, along with the key expansion process. It also explores implementation aspects, including the use of look-up tables, and examines two modes of operation. The end of the section focuses on the main hardware components of the PROACT chip, providing information on the RISC-V Instruction Set Architecture and the Ibex cores that are used in the PROACT chip.

3.1 The Advanced Encryption Standard

The Advanced Encryption Standard, as described in the FIPS publication [NIS01a], is a symmetric block cipher that operates on blocks with a size of sixteen bytes, which are structured as a two-dimensional, column-major array with four rows. This array is denoted as the *state array* or solely the *state* (see Figure 1). The size of the key can be specified to be 128, 192, or 256 bits, where a larger key size offers a higher level of security. The process of expanding the key into different round keys is called *key expansion* and is explained in Section 3.1.1. The number of rounds (Nr) of the cipher is dependent on the size of the key, where for key sizes of 128, 192, and 256 bits, the number of rounds is 10, 12, and 14, respectively. At the start of each encryption process, the first round key is added to the state (`AddRoundKey`). After this, four mathematical operations are performed on the state in each round: `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. These mathematical operations treat the bytes in the state as polynomials in the field $GF(2^8)$. Knowledge of this field is not necessary to understand the overall behaviour of the algorithm and is therefore not discussed in detail. The key point is that addition and multiplication operate differently for numbers in this field.

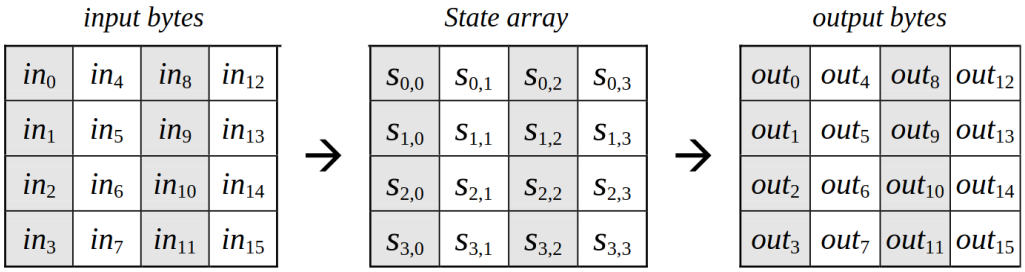

Figure 1: The plaintext input of 16 bytes is mapped into a four-by-four array called the state [NIS01a].

Listing 1 shows the pseudocode for the encryption of one block of plaintext. The four round operations, that are represented as function calls in the pseudocode, and their inverses are specified as follows:

1. `SubBytes`: each byte is substituted by a byte specified in the substitution table (S-box). This operation can be inverted by using the inverse of the S-box as the substitution table.

2. `ShiftRows`: the three bottom rows of the state array are cyclically shifted to the left by a specified offset, where the offset is equal to the row number. This means that the top row

(denoted as row 0) is not shifted, row 1 is cyclically shifted with one byte to the left, etc. This operation can be inversed by shifting the rows to the right with the same offset.

3. `MixColumns`: each column is multiplied by a fixed polynomial in the field of $GF(2^8)$. This multiplication can be described as a matrix multiplication, where $S_{i,j}$ are the bytes of a column j in the state and $S'_{i,j}$ are the bytes of the same column in the state after the matrix multiplication is performed:

$$\begin{bmatrix} S'_{0,j} \\ S'_{1,j} \\ S'_{2,j} \\ S'_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} S_{0,j} \\ S_{1,j} \\ S_{2,j} \\ S_{3,j} \end{bmatrix} \tag{1}$$

Note that after this transformation, each byte $S'_{i,j}$ of column j will consist of a combination of bytes $S_{0,j}$, $S_{1,j}$, $S_{2,j}$ and $S_{3,j}$ of column j before the transformation. The `MixColumns` operation is left out in the last round. The inverse of this operation is similar, but the matrix shown in Equation (2) is used for the multiplication. Note that the entries in this matrix are hexadecimal values.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \tag{2}$$

4. `AddRoundKey`: In the field of $GF(2^8)$, addition is specified as a bitwise exclusive-or operation. Therefore, the `AddRoundKey` operation performs a bitwise XOR with the state and the round key. Since an XOR operation can be inversed by performing the same XOR operation again, the inverse for the `AddRoundKey` operation is identical to the regular operation.

```
1  Cipher(State, RoundKey) {
2      AddRoundKey(State, RoundKey, 0);
3
4      for (i = 1; i < Nr; i += 1) {
5          SubBytes(State);
6          ShiftRows(State);
7          MixColumns(State);
8          AddRoundKey(State, RoundKey, i);
9      }
10
11     SubBytes(State);
12     ShiftRows(State);
13     AddRoundKey(State, RoundKey, Nr);
14 }
```

Listing 1: Pseudocode for the encryption of one block of plaintext.

3.1.1 Key expansion

The process of generating $Nr + 1$ round keys from the original cipher key is called *key expansion*. Let `RoundKey` be the array of words in which the expanded key is stored. Let Nk denote the number of words in the original key, so $Nk = 4$ for a key size of 128 bits. The formula below shows how the word $\texttt{RoundKey}_i$ is generated. In this formula, `Sub` is the per-byte substitution of the word with the values stored in the S-box. `Rotate` is a cyclic left-shift of the bytes in a word. Lastly, `Rcon` denotes an array of round constants. It can be seen that the key for the first Nk words is the same as the cipher key. After that, the words in the round key are built from previous words.

$$\texttt{RoundKey}_i = \begin{cases} \texttt{Key}_i & i < Nk \\ \texttt{RoundKey}_{i-Nk} \oplus & \\ \quad \texttt{Sub}(\texttt{Rotate}(\texttt{RoundKey}_{i-1}) \oplus \texttt{Rcon}[i/Nk]) & i \geq Nk, i \equiv 0 \pmod{Nk} \\ \texttt{RoundKey}_{i-Nk} \oplus \texttt{Sub}(\texttt{RoundKey}_{i-1}) & i \geq Nk, Nk = 6, i \equiv 4 \pmod{Nk} \\ \texttt{RoundKey}_{i-Nk} \oplus \texttt{RoundKey}_{i-1} & \text{otherwise} \end{cases}$$

3.1.2 Decryption

The cipher can be inversed by applying the inverse of the four round operations in the opposite order (`AddRoundKey`, `InvMixColumns`, `InvShiftRows`, `InvSubBytes`). Listing 2 shows the pseudocode for this straightforward method of decryption. Note that `InvMixColumns` is now absent in the first round and an additional `AddRoundKey` operation is performed after the last round. Both the cipher and inverse cipher use the same round keys but the inverse cipher applies them in opposite order.

```
1  InvCipher(State, RoundKey) {
2      AddRoundKey(State, RoundKey, Nr);
3      InvShiftRows(State);
4      InvSubBytes(State);
5
6      for (i = Nr - 1; i > 0; i -= 1) {
7          AddRoundKey(State, RoundKey, i);
8          InvMixColumns(State);
9          InvShiftRows(State);
10         InvSubBytes(State);
11     }
12
13     AddRoundKey(State, RoundKey, 0);
14 }
```

Listing 2: Pseudocode for the straightforward decryption of one block of ciphertext.

Apart from this straightforward method of decryption, the standard specifies an equivalent decryption method which preserves the order of the round operations from the encryption algorithm. Preserving the general structure of the cipher can be beneficial for optimizing the implementation. For example, this structure is needed to implement decryption using look-up tables (see

Section 3.1.3). It should be clear that the order of `InvShiftRows` and `InvSubBytes` can be reversed, since `InvSubBytes` operates on individual bytes and `InvShiftRows` does not affect individual bytes. The operations `InvMixColumns` and `AddRoundKey` can also be reversed if the round keys are changed accordingly. `InvMixColumns` is a linear operation and for any linear operation f, it holds that $f(x \oplus y) = f(x) \oplus f(y)$. Thus, the order of the two operations can be reversed if `InvMixColumns` is applied to each of the round keys, except for the first and the last one. The restructured equivalent decryption algorithm is shown in Listing 3. Notice that apart from rearranging the order of the operations, the first `InvShiftRows` and `InvSubBytes` operations have been moved inside the loop and the last `InvShiftRows` and `InvSubBytes` have been moved outside the loop so that the structure of the rounds and final round for the decryption algorithm is identical to the structure of the rounds and final round of the encryption algorithm.

```
1  EqInvCipher(State, EqRoundKey) {
2      AddRoundKey(State, EqRoundKey, Nr);
3
4      for (i = Nr - 1; i > 0; i -= 1) {
5          InvSubBytes(State);
6          InvShiftRows(State);
7          InvMixColumns(State);
8          AddRoundKey(State, EqRoundKey, i);
9      }
10
11     InvSubBytes(State);
12     InvShiftRows(State);
13     AddRoundKey(State, EqRoundKey, Nr);
14 }
```

Listing 3: Pseudocode for the decryption of one block of ciphertext using the equivalent decryption method.

3.1.3 Implementation using look-up tables

The designers of the block cipher describe an efficient implementation of AES on 32-bit platforms [DR02, Ch. 4.2]. Research has shown that table-based implementations are generally vulnerable to cache-based timing attacks [Ber05, BM06], where the key can be inferred by analyzing the timing delays of accessing table elements that are stored in different cache levels. However, the PROACT chip does not have a cache and is therefore assumed to be immune to attacks based on this vulnerability.

The table-based implementation significantly reduces execution time by combining the `SubBytes` and `MixColumns` steps into look-up table operations. Using basic linear algebra, the matrix multiplication for a single column shown in Equation (1) can be written as a linear combination of four vectors. Since the matrix is fixed, each of these vectors can be precomputed for all 256 possibilities of the coefficients after the `SubBytes` operation is performed on the coefficients. This creates four look-up tables (T_0, T_1, T_2 and T_3) with 256 word-sized entries each.

$$T_0[a] = \begin{bmatrix} 02 \cdot sbox[a] \\ 01 \cdot sbox[a] \\ 01 \cdot sbox[a] \\ 03 \cdot sbox[a] \end{bmatrix}, \; T_1[a] = \begin{bmatrix} 03 \cdot sbox[a] \\ 02 \cdot sbox[a] \\ 01 \cdot sbox[a] \\ 01 \cdot sbox[a] \end{bmatrix}, \; T_2[a] = \begin{bmatrix} 01 \cdot sbox[a] \\ 03 \cdot sbox[a] \\ 02 \cdot sbox[a] \\ 01 \cdot sbox[a] \end{bmatrix}, \; T_3[a] = \begin{bmatrix} 01 \cdot sbox[a] \\ 01 \cdot sbox[a] \\ 03 \cdot sbox[a] \\ 02 \cdot sbox[a] \end{bmatrix} \tag{3}$$

The computation of the `SubBytes` and `MixColumns` step for one column of the state can now be written as follows:

$$\begin{bmatrix} S'_{0,j} \\ S'_{1,j} \\ S'_{2,j} \\ S'_{3,j} \end{bmatrix} = T_0[S_{0,j}] \oplus T_1[S_{1,j}] \oplus T_2[S_{2,j}] \oplus T_3[S_{3,j}] \tag{4}$$

where $S_{0,j} \ldots S_{3,j}$ are the bytes of column j in the old state after `ShiftRows` has been performed and $S'_{0,j} \ldots S'_{3_j}$ denotes the same column j in the new state.

The decryption look-up tables can be computed and applied in the same manner as the encryption look-up tables if the equivalent decryption algorithm described in Section 3.1.2 is used. This results in a combined size of 8 KB for the encryption and decryption look-up tables.

3.1.4 Modes of operation

The AES algorithm is a block cipher and thus only specifies how a single block of plaintext is encrypted. A mode of operation defines how subsequent blocks of plaintext are processed such that it is possible to encrypt data with a size larger than sixteen bytes. Five modes of operation are recommended and described by the National Institute of Standards and Technology (NIST) to be used in combination with AES [Dwo01]. This section briefly discusses the two modes of operation that are relevant to this research.

In Cipher Block Chaining Mode (CBC), each block of plaintext is XOR-ed with the previous block of ciphertext before it is encrypted. The first block of plaintext is XOR-ed with an Initialization Vector (IV). The IV should be unpredictable, which can be achieved by initializing the IV randomly. For decryption in this mode, the AES inverse is applied to each ciphertext block and the output is XOR-ed with the previous ciphertext block to obtain the plaintext block. The first block is XOR-ed with the IV after the inverse of AES has been applied. Figure 2 shows a visual representation of this mode. To use this mode of operation, the input must be padded to a multiple of the block size.

The second mode of operation that is relevant to this research is the Counter Mode (CTR). In this mode, a counter sequence $Counter_1 \ldots Counter_n$ is chosen. The sequence of natural numbers is often used for this purpose. Each counter is encrypted by the AES block cipher and produces one output block. This output block is then XOR-ed with the plaintext block to produce the ciphertext. Decryption is done in the same way, except that the encrypted counter is now XOR-ed with the ciphertext block to obtain the plaintext block. See Figure 3 for a visual representation of this mode.

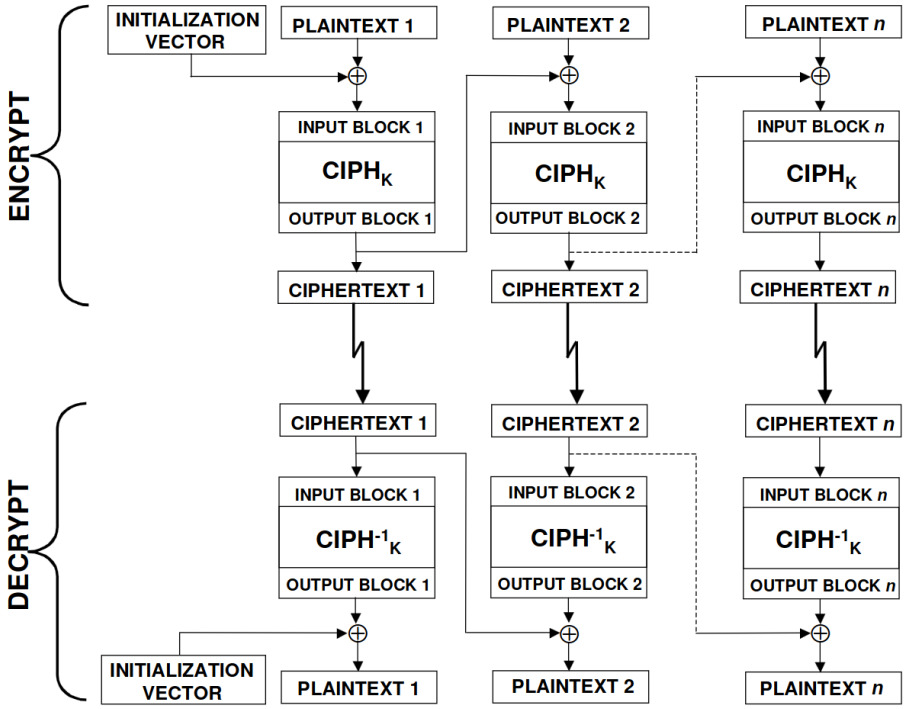

Figure 2: The CBC mode [Dwo01]

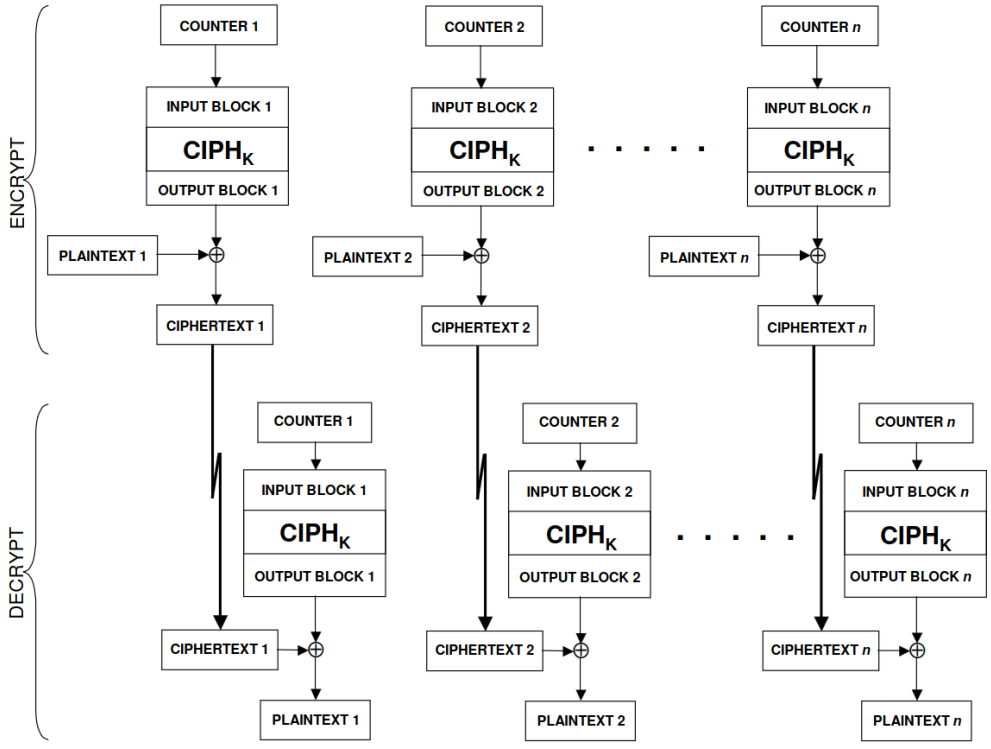

Figure 3: The CTR mode [Dwo01]

3.2 The RISC-V ISA

The PROACT chip features two cores that employ the open-source, royalty-free RISC-V [RIS24] Instruction Set Architecture (ISA). As indicated by the name, this ISA is based on the Reduced Instruction Set Computer (RISC) principles, that are characterized by simple instructions to minimize execution time. The RISC-V ISA is a load/store architecture and has, in its basic implementation, 32 general-purpose registers. It uses little-endian byte ordering, which means that the least significant byte of a word is stored at the lowest memory address. There are different variants of the instruction set for 32-bit and 64-bit address spaces but since the Ibex cores that are used in the PROACT chip are 32-bit cores, only the former variant is considered.

The RISC-V ISA defines the Base Integer Instruction Set (RV32I) as the minimum instruction set that should be supported by hardware implementing the RISC-V architecture. This instruction set features basic arithmetic and bitwise instructions in two variants: register-register and register-immediate instructions. The latter category is indicated by placing an 'i' after the instructions (e.g. `add` becomes `addi`). Furthermore, the RV32I features control transfer instructions and instructions to load and store words, half words and bytes. In addition to this Base Integer Instruction Set, hardware may support one or multiple extensions. The PROACT chip supports the Standard Extension for Integer Multiplication and Division (M) and the Standard Extension for Compressed Instructions (C). Compressed instructions have a length of 16 bits instead of the normal 32 bits and can therefore reduce code size.

3.3 Main PROACT chip hardware components

This thesis focuses on optimizing the AES algorithm for the PROACT chip. The main components of the PROACT chip are two Ibex [ETH20] cores, which are 32-bit open-source RISC-V cores. Figure 4 shows a block diagram of the Ibex core. These cores are well suited for embedded systems

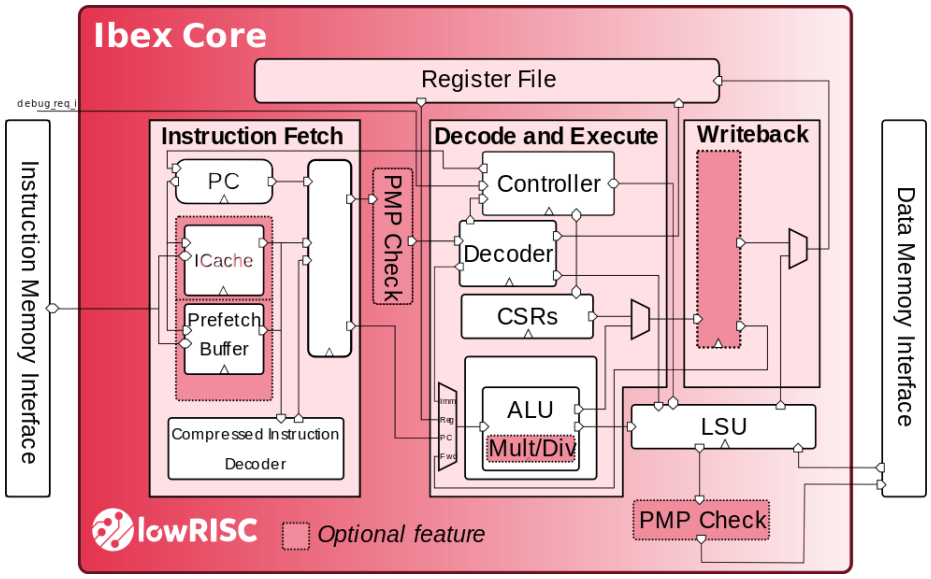

Figure 4: Block diagram of the Ibex core [ETH20]

due to their simple design and highly parameterizable nature. One of the two cores is dedicated to running the cryptographic algorithms, while the second one acts as a controller. Both cores on the PROACT chip prototype have separate 64 KB data and 64 KB instruction memories. The address space theoretically allows each memory section to be extended to 32 MB, except for the instruction memory of the cryptographic core, which can be extended to 64 MB [Saj23]. This thesis is concerned with the core that runs the cryptographic algorithms, with an instruction and data memory size of 64 KB each.

As previously stated, the Ibex core is customizable in various ways. The cores that are used in the PROACT chip support the Base Integer Instruction Set (RV32I) and are configured to support the Integer Multiplication and Division (M) and Compressed (C) instruction extensions. The writeback stage, branch predictor and physical memory protection (PMP) are disabled. The prefetch buffer (using a FIFO algorithm) is enabled. Since the prefetch buffer and the instruction cache can not be enabled at the same time, the instruction cache is disabled. The execute stage features a fast multi-cycle multiplier, which means that the execution of an integer multiplication instruction requires three clock cycles to complete. The integer instructions within the base instruction set execute within a single clock cycle, whereas aligned load and store instructions on the PROACT chip require two clock cycles to complete. Misaligned memory accesses are supported but introduce an additional stall cycle.

4 Methodology

The aim of this research is to develop optimized AES encryption and decryption implementations for the PROACT chip with the CBC and CTR modes of operation. The optimization objective is the reduction of clock cycles required to execute these algorithms. This section provides a detailed explanation of the approach undertaken to achieve this objective.

4.1 Overview

An optimized implementation for encrypting a single block of plaintext is written in C, using the same algorithmic approach as the assembly implementation that is developed and optimized by Stoffelen [Sto19], while maintaining readability and extensibility. The performance in clock cycles of the C implementation is compared to the performance of the assembly implementation. The assembly code generated from the C code by the compiler is examined and compared to the RISC-V assembly implementation to ascertain how the C code can be improved. This process of measuring the performance of the C implementation and comparing the generated assembly code to the handwritten assembly code continues until the performance of the C implementation is comparable to that of the assembly implementation, or until it is clear that this objective can not be reached. In this study, 'comparable performance' is defined as a margin of 5% increase in the number of clock cycles compared to the number of clock cycles required by the assembly implementation executing the same functionality. If comparable performance can be achieved, the key expansion, decryption and modes of operation are be implemented in C. If this is not possible, the same components are implemented in RISC-V assembly. To optimize AES in CTR mode, optimization techniques proposed by Park and Lee [PL18] are employed.

4.2 Single-block encryption implementation

An AES encryption implementation featuring a key size of 128 bits is written in C to approximate Stoffelens assembly code [Sto19], while still maintaining the benefits that are offered by using a high-level language. Both implementations use the same algorithmic approach to AES and both use identical look-up tables. The function that encrypts one block of plaintext is called `encrypt`

```
1  static inline void encrypt(const uint8_t *roundKey, const uint8_t *in,
2      uint8_t *out) {
3      for (int i = 0; i < 4; i++)
4          ((uint32_t *)out)[i] = ((uint32_t *)in)[i] ^ ((uint32_t *)roundKey)[i];
5
6      for (int round = 1; round < Nr; round++)
7          encrypt_round(out, roundKey, round);
8
9      encrypt_final_round(out, roundKey);
10 }
```

Listing 4: C-code for the `encrypt` function.

and is shown in Listing 4. The contents of the `roundKey` array should be computed by the function that performs the key expansion before the encrypt function is called. The parameters `in` and `out` correspond to pointers to the input and output arrays respectively. The assembly code loads four bytes from the input array into the same register at once. In C, this is emulated by casting the `uint8_t` pointer to a `uint32_t` pointer before it is dereferenced, as shown in Listing 4, line 4. Note that each of the four words loaded from the input in this manner corresponds to a column of the state.

The function `encrypt` adds the first round key to the input and stores it in the output, as can be seen in Listing 4, lines 3–4. Subsequently, the function enters a loop with $Nr-1$ iterations to execute the encryption rounds (Listing 4, lines 6–7). In the loop, the `out` pointer is passed to the function `encrypt_round(uint8_t *state, const uint8_t *roundKey, uint32_t round)`, where `round` denotes the number of the round. Most of the computational work of the encryption algorithm is handled within this function. Listing 5 shows the pseudocode for it. In reality, the inner loop of the `encrypt_round` function (lines 6–18) is unrolled and the array `tempa` declared in line 2 is replaced by four distinct variables to improve performance. The castings from `uint8_t` pointers to `uint32_t` pointers for the `state` and `roundKey` parameters are omitted in the pseudocode. All

```
1  static inline void encrypt_round(state, roundKey, round) {
2      tempa[4] = {roundKey[4 * round], roundKey[4 * round + 1],
3          roundKey[4 * round + 2], roundKey[4 * round + 3]};
4
5      for (int i = 0; i < 4; i += 1) {
6          for (int j = 0; j < 4; j += 1) {
7              // Shift rows with offset i.
8              // The rows are shifted to the right because RISC-V is little-endian.
9              temp = state[(j + i) % 4];
10
11             // Extract byte i from the column.
12             // This is the index into the look-up table.
13             LUT_i = (temp >> (i * 8)) & 0xff;
14
15             // Index into the look-up table and XOR this value with the previous
16             // result. Note that this is the round key in the first iteration.
17             tempa[j] ^= TableEncrypt[(LUT_i * 4) + i];
18         }
19     }
20
21     // Store the result back into the state
22     for (int i = 0; i < 4; i++) {
23         state[i] = tempa[i];
24     }
25 }
```

Listing 5: Pseudocode for the `encrypt_round` function.

variables and arrays, except for the loop variables, should be interpreted as variables and arrays of the type `uint32_t`. The four look-up tables as described in Equation (3) are combined into one large look-up table (`TableEncrypt`), where the elements with the same indices in different tables are stored adjacent to each other, in the following order: $T_2[0]$, $T_0[0]$, $T_1[0]$, $T_3[0]$, ..., $T_2[255]$, $T_0[255]$, $T_1[255]$, $T_3[255]$. By positioning $T_2[i]$ as the first word in each group of four words, the first byte of each group of sixteen bytes can be readily extracted to form the S-box. The actual `encrypt_round` function features logic to add the correct offset to the look-up table index instead of adding the loop index i.

Equation (4) shows how the `SubBytes` and `MixColumns` operations can be combined using look-up tables. While the formula implies separate calculations for each column, it is evident that an alternative approach is also possible: computing all values that use T_0 first, followed by those using T_1, and so forth, with the intermediate results stored in four temporary variables. This is the approach taken in the assembly implementation and the same approach is employed in the C implementation. The `AddRoundKey` operation is performed by XOR-ing the results of the first iteration of the outer for-loop with the round key for the corresponding round, as shown in Listing 5, line 17. Once the computation for one round is completed, the temporary variables are stored back into the state. This operation is performed in Listing 5, lines 22–24.

```
1  static inline void encrypt_final_round(state, roundKey, round) {
2      tempa[4] = {roundKey[4 * round], roundKey[4 * round + 1],
3          roundKey[4 * round + 2], roundKey[4 * round + 3]};
4
5      for (int i = 0; i < 4; i += 1) {
6          for (int j = 0; j < 4; j += 1) {
7              // Shift rows with offset i.
8              temp = state[(j + i) % 4];
9
10             // Extract byte i from the column. This is the index into the S-box.
11             LUT_i = (temp >> (i * 8)) & 0xff;
12
13             // Retrieve the S-box value and shift the extracted byte to the
14             // correct position. XOR the value with the previous result.
15             tempa[j] ^= (TableEncrypt[LUT_i * 4] & 0xff) << (i * 8);
16         }
17     }
18
19     // Store the result back into the state
20     for (int i = 0; i < 4; i++) {
21         state[i] = tempa[i];
22     }
23 }
```

Listing 6: Pseudocode for the `encrypt_final_round` function.

After the first $Nr-1$ rounds have been completed inside the loop, the function `encrypt_final_round` is called to perform the final round (Listing 4, line 9). The pseudocode `encrypt_final_round` is shown in Listing 6. Recall that this round is the same as all previous rounds, except for the absence of the `MixColumns` operation. Therefore, the implementation is similar to the `encrypt_round` function, except that the S-box instead of the look-up table is indexed with the computed value of `LUT_i`, as shown in line 15. This line also shows that an additional AND and shift instruction is needed to extract the first byte of the word-sized value obtained from the look-up table and to shift this value back to the correct place in the state array. Because the AND instruction 'masks' the right-most byte in the register, this operation is also referred to as a mask operation.

While the C implementation is developed to approximate the assembly implementation, some differences are present to maintain a high-level language approach.

- All loops are unrolled in the assembly implementation. However, the C implementation uses loops to perform the first `AddRoundKey` operation and to iterate over the encryption rounds.
- In the assembly implementation, the code that performs the round operations for each of the four rows is nearly identical. These computations are combined into a loop in the C implementation, which is the outer loop in Listing 5. Due to the use of this loop, some extra computations need to be performed. For example, to extract the correct byte of a column to compute the look-up table index, the 32-bit value of the column needs to be shifted $i \times 8$ bits to the right. This multiplication is not present in the assembly code, as the offset can be hardcoded when the loop is unrolled.
- The code that executes one encryption round is implemented in an inline function, while this is a macro in the assembly code. This functionality is not implemented as a macro in C, because an inline function still eliminates the function call overhead while maintaining benefits like type-checking.
- Initially, shift operations were used in the C code for all occurrences of shift instructions in the assembly code. However, this degraded the performance in some instances because the compiler was not able to combine two subsequent shift instructions into one. When multiplication is used in those instances, the compiler can combine the generated shift instructions with other shift instructions, leading to a better performance. Thus, multiplication is used instead of shift operations in those cases.

4.3 Complete AES implementation

The choice of the language (C or assembly) in which the key expansion, decryption and modes of operation are implemented is dependent on the performance of the C implementation of the encryption algorithm. In the case that the performance of the encryption implementation in C is comparable (with a margin of 5%) to that of the assembly implementation, the AES implementation is completed and optimized in C. Otherwise, the same is done using the RISC-V assembly.

4.3.1 Key expansion

The encryption algorithm's key expansion is already present in the assembly code. If the AES implementation is to be completed in C, the same approach as with the encryption algorithm is applied. This means that the code is developed using the same algorithmic procedure employed in the assembly key expansion implementation. Subsequently, the compiled C code is analyzed to enhance its performance until it is comparable to that of the assembly code.

In Section 3.1.2, it is discussed that an adapted key expansion algorithm is required for decryption when a table-based approach is used to implement AES. This algorithm is referred to as the decryption or equivalent key expansion algorithm. The algorithm begins by executing the standard key expansion process with the encryption key. Subsequently, all round keys, except for the first and last one, are transformed by applying the `MixColumns` operation. Regardless of the choice of language for the implementation of the key expansion, the `MixColumns` operation can be performed similarly as it is performed in the encryption algorithm, but utilizing the decryption look-up table instead of the encryption look-up table. An additional S-box look-up operation is also needed to reverse the effect of the inverse S-box that is present in the decryption look-up table.

4.3.2 Decryption

Due to the use of the equivalent decryption algorithm, the implementation of the decryption is straightforward. However, there are two notable distinctions: the round keys must be applied in inverse order, and during the `InvShiftRows` operation, the rows are shifted to the right instead of the left. Since the algorithm is not fundamentally different, our choice is to adopt the optimized version of the encryption algorithm without any further optimizations. Note that an extra table with 256 byte-sized entries is required to store the values of the inverse S-box for the final round because the inverse S-box can not be directly extracted from the decryption look-up table. This will presumably result in a slightly improved performance in the last round compared to the encryption algorithm because the mask (AND) operation that is needed in the encryption final round to obtain the S-box value from the look-up table is eliminated.

4.3.3 CBC mode

The implementation of cipher block chaining is relatively straightforward, as it primarily revolves around additional XOR operations. The focus in optimizing this mode therefore lies in reducing the number of assembly instructions. Programming this mode of operation directly in assembly would facilitate precise control to minimize the number of instructions. If a C implementation is chosen, the compiled C code is examined to identify redundant instructions and the C code is modified to eliminate these instructions.

4.3.4 CTR mode

The second mode of operation that is implemented is counter mode. To optimize this mode, some of the techniques mentioned by Park and Lee in [PL18] are applied. Recall that in CTR mode, a counter value is encrypted and XOR-ed with the block of plaintext. The counter value is increased by one for the next block of plaintext. In $2^8 - 1$ of the cases, this increase only affects the last

byte of the counter, thereby creating the opportunity to reuse certain results from the previous encryption of the counter. Park and Lee propose five optimizations based on this observation. Three of the proposed optimizations are implemented and tested for performance gain. The two other techniques that are described in the paper are only beneficial if the input size exceeds 256 blocks (4 KB) and the speedup that is achieved with these additional optimizations is relatively small. Therefore, our choice is to focus solely on the other three optimizations. These optimizations are:

- FACE$_{rd0}$: This technique optimizes the first `AddRoundKey` operation that is performed at the start of the process of encrypting each block of plaintext. The first three columns of the result state are cached and reused. This means that in most cases, only one XOR operation is needed to compute the initial `AddRoundKey` operation.
- FACE$_{rd1}$: This optimization utilizes the fact that a change in the last byte of the CTR value only affects the last column of the state in the first round. Therefore, the other three columns can be cached and reused in the first round.
- FACE$_{rd2}$: The input block of round two is equal to the output block of round one. Therefore, the subsequent input blocks of round two only differ from each other in the last column in most cases. After the `ShiftRows` operation is performed, each column of the state contains exactly one byte that is different from the previous block computation. This difference will have spread throughout the whole state by the end of round two. However, intermediate calculation results of the `MixColumns` and `AddRoundKey` operations can still be cached and reused since three of the four bytes in each column are identical to those of the previous block after the `ShiftRows` operation.

4.3.5 Expansion to larger key sizes

Regardless of whether the C implementation reaches comparable performance to the assembly code, it is expected that the C implementation still has a slightly lower performance. Should the AES implementation be completed in C, the algorithm is extended to support key sizes of 192 and 256 bits to demonstrate the advantage of the flexibility that is provided by programming in a higher-level language. These larger key sizes provide a higher level of security compared to a key size of 128 bits.

4.4 Compilation

Cross-compilation for RISC-V is achieved using the rv32imc GCC toolchain. The RISC-V ISA that is used in the PROACT chip (RV32IMC) can be configured with the compilation options `-march=rv32imc -mabi=ilp32`. All code is compiled with the optimization option `-O3` enabled to optimize for execution speed.

4.5 Experimental setup

During the process of developing the optimized AES implementations, the performance is measured in clock cycles with Simple System[1], which uses Verilator to emulate an Ibex based system. This

[1]See https://github.com/lowRISC/ibex/tree/master/examples/simple_system for more information about Simple System.

emulator is also used to verify the correctness of the implementations against the AES test vectors provided by NIST [NIS01a]. Simple System can be configured to use different Ibex configurations. The provided 'simple' configuration matches the configuration of the Ibex cores that are integrated into the PROACT chip. Simple System provides functions to reset, enable and disable the performance counters. Measuring the performance of a function `encrypt` can be done as shown in Listing 7. The statements to enable and disable the performance counters are placed around the function call in all experiments. This means that the overhead of the function call is also measured, which is a realistic scenario for the use of external functions.

```
int main() {
    pcount_enable(0);
    pcount_reset();

    // Some code ...

    pcount_enable(1);
    encrypt();
    pcount_enable(0);

    // Some more code ...

    return 0;
}
```

Listing 7: Example of performance measurement with Simple System.

Once the process of developing the optimized AES implementations is completed, the performance results and cryptographic correctness of the implementations are verified on the PROACT chip prototype using the PYNQ-Z2 FPGA development board [Xil20]. A graphical user interface is developed to enhance the efficiency of this process and to aid further research and development on the PROACT chip. The implementation of this graphical user interface is based on the implementation of a basic user interface that has already been developed for this chip [Saj23]. However, the previous interface requires the user to have an understanding of the specific design of the PROACT chip, whereas the new interface enables the user to supply their encryption algorithm written in C or RISC-V assembly and test this implementation on the chip without specifying any additional details. A user manual for the new graphical user interface can be found in Appendix A.

5 Experimental Results

The results of the analysis of the instructions present in the compiled C code of the AES encryption implementation are presented in Section 5.1. Optimizations based on this analysis are explained along with the performance results. The final version of the C encryption implementation reaches a comparable performance to the assembly implementation. Thus, the AES implementation is completed in C. Implementation details and performance results for the completed AES implementation are provided in Section 5.2. Lastly, Section 5.3 presents the results that are obtained on the PROACT chip prototype.

5.1 Single-block encryption

Single-block encryption is implemented in C as explained in Section 4.2. This implementation is referred to as the C implementation with optimization level 0 (opt 0). Based on the insights gained in the analysis of this implementation, a new version of the encryption algorithm has been written, which is referred to as the implementation with optimization level 1 (opt 1). A final version, where the main encryption loop is unrolled, has also been implemented. This version is denoted as the implementation with optimization level 2 (opt 2). Table 1 displays a summary of the performance results on the Ibex emulator.

	Assembly	C (opt 0)	C (opt 1)	C (opt 2)
Clock cycles (absolute)	1121	1384	1189	1149
Clock cycles (relative to [Sto19])	0%	+23%	+6.1%	+2.5%

Table 1: Performance in clock cycles of single-block encryption, measured on the Ibex emulator.

5.1.1 First version

The C implementation with optimization level 0 is approximately 23% slower than the assembly implementation. The number of instructions and the instruction types are examined to explain this difference. Any instruction denoted as the type ‘int‘ requires one clock cycle to execute, as well as any branch instruction that is not taken. All load and store instructions require two clock cycles to complete. Lastly, instructions for taken branches require three clock cycles to execute.

A general examination of the compiled C code shows that all loops, except for the main loop that iterates over the rounds, are unrolled by the compiler. The compiled C code reserves more space on the stack to store callee-saved registers than the assembly code does. Table 2 shows how the instructions are distributed over the code's functionalities. The largest difference in the number of instructions can be found within the main loop, for which a closer examination is needed.

5.1.2 Analysis of instructions per round

The compiler generates twelve additional integer instructions per encryption round to compute the addresses for the table look-up operation compared to the assembly code. In Listing 5, the outer loop can be seen as iterating over the rows (0 up to 3) of the state, while the inner loop

Instruction purpose	Type	Assembly	C (opt 0)	Total difference
Stack	sw/lw	12	22	+10
	int	2	2	0
Loading / creating constants	int	7	5	−2
First `AddRoundKey`	lw	8	8	0
	int	4	4	0
	sw	0	4	+4
Loop initialization	int	0	2	+2
Encryption round	lw/sw	20	24	+36
(per round)	int	64	76	+108
Loop increment and compare	int	0	1	+9
(per round)	branch	0	1	+9
Final round	lw	4	16	+12
	lbu	16	4	−12
	sw	4	4	0
	int	76	96	+20

Table 2: The number and types of instructions for the initial single-block encryption implementation compared to the assembly implementation. In the last column, the number of instructions executed for encrypting one block in the C implementation is compared to the number of instructions executed for the same task in the assembly implementation.

iterates over the columns (0 up to 3) of the state. The look-up table addresses for rows 0 and 3 are computed with three instructions in both the assembly and C code.

For the computation of the addresses for row 1, the compiled C code uses two extra instructions per column. The computation is shown in Listing 8. Each column is right-shifted with six bits (line 4) and masked with the value `0x3fc` (line 5) to extract the second byte of the original column. Due to the structure of the combined look-up table, the value 3 is added to this result (line 6). Lastly, the value is shifted to the left with two bits (line 7) to compute the table offset in bytes. In the assembly code, the same result is achieved by right-shifting the column with four bits and masking the result with `0xff0`, as can be seen in Listing 9. There is no need to add the value 3 to

```
1    ; Pre: a1 contains column 1 of the state.
2    ; Pre: LUT contains the address of the combined look-up table.
3    ; Post: t4 contains the computed address.
4    srli    t0, a1, 6
5    andi    t0, t0, 0x3fc
6    addi    t0, t0, 3
7    slli    t0, t0, 2
8    add     t4, t0, \LUT
```

Listing 8: Look-up table address computation for row 1 (column 0) in the `encrypt_round` function of the *generated* assembly code.

```
1     ; Pre: a1 contains column 1 of the state.
2     ; Pre: LUT3 contains the address of the fourth word in the combined look-up
3     ; table.
4     ; Post: t4 contains the computed address.
5     srli    t0, a1, 4
6     andi    t0, t0, 0xff0
7     add     t4, t0, \LUT3
```

Listing 9: Look-up table address computation for row 1 (column 0) in the `encrypt_round` function of the *written* assembly code.

index the correct look-up table because the addresses to the first four words in the look-up table are stored in registers and can therefore be directly accessed. Contrarily, the compiler does not automatically store these addresses in registers and it also fails to combine the two shift instructions.

The compiler generates one extra instruction per column for the computation of the look-up table addresses for row 2. Listing 10 shows this computation. Each column is right-shifted with sixteen bits (line 5), the lowest byte is extracted with a mask (line 6) and then left-shifted with four bits to compute the table offset (line 7). In the assembly code, the same result is achieved by right-shifting the column with twelve bits and masking the result with the value `0xff0`, as shown in Listing 11.

```
1     ; Pre: a2 contains column 2 of the state.
2     ; Pre: LUT contains the address of (the first word of) the combined look-up
3     table.
4     ; Post: t4 contains the computed address.
5     srli    t0, a2, 16
6     andi    t0, t0, 0xff
7     slli    t0, t0, 4
8     add     t4, t0, \LUT
```

Listing 10: Look-up table address computation for row 2 (column 0) in the `encrypt_round` function of the *generated* assembly code.

```
1     ; Pre: a2 contains column 2 of the state.
2     ; Pre: LUT0 contains the address of the first word in the combined look-up
3     ; table.
4     ; Post: t4 contains the computed address.
5     srli    t0, a2, 12
6     andi    t0, t0, 0xff0
7     add     t4, t0, \LUT0
```

Listing 11: Look-up table address computation for row 2 (column 0) in the `encrypt_round` function of the *written* assembly code.

Apart from the extra integer instructions that are generated for each round, the compiler generates four extra store instructions to store the new state at the end of each round. This is also the

case after the first `AddRoundKey` operation. These instructions are unnecessary, as computation continues in the new round with the same register values that were just stored, and the registers are once again stored in the same locations.

5.1.3 Analysis of instructions in the final round

Table 2 shows that the compiled C code contains twenty additional integer instructions to complete the final round. As is the case with the rounds of the main loop, the number of additional instructions differs per row. The number of instructions in the compiled C code and the assembly code is the same for the computation of the values in row 3.

One additional shift instruction per column is generated by the compiler for the computation of the values for row 0. The code for the computation of `ShiftRows` and `SubBytes` for a column of row 0 can be seen in Listing 12. Note that the `ShiftRows` operation does not have any effect on row 0. The compiled C code shifts the column value to the left with two bits (line 4), masks the value with `0x1fe` to extract the last byte of the column (line 5) and shifts the value to the left again to compute the table offset in bytes (line 6). Note that the value is not first masked and then shifted to the left with four bits, even though the compiler does generate this version for the `encrypt_round` function. This can be explained by the fact that code within loops is generally subject to stricter optimization compared to code outside of loops. The assembly code combines the two shift instructions, as shown in Listing 13. Both the assembly and the C implementations use the `lbu` (load byte unsigned) instruction to eliminate the need for any masking that would otherwise be required to extract the correct byte from the obtained word.

```
1    ; Pre: a0 contains column 0 of the state.
2    ; Pre: LUT contains the address of the combined look-up table.
3    ; Post: t0 contains the obtained S-box value.
4    slli    t0, a0, 2
5    andi    t0, t0, 0x1fe
6    slli    t0, t0, 2
7    add     t4, t0, \LUT
8    lbu     t0, (t4)
```

Listing 12: computation of (`ShiftRows` and) `SubBytes` for row 0 (column 0) in the `encrypt_final_round` function of the *generated* assembly code.

```
1    ; Pre: a0 contains column 0 of the state.
2    ; Pre: LUT contains the address of the combined look-up table.
3    ; Post: t0 contains the obtained S-box value.
4    slli    t0, a0, 4
5    andi    t0, t0, 0xff0
6    add     t4, t0, \LUT
7    lbu     t0, (t4)
```

Listing 13: computation of (`ShiftRows` and) `SubBytes` for row 0 (column 0) in the `encrypt_final_round` function of the *written* assembly code.

To compute the look-up table addresses for rows 1 and 2, the compiled C code follows the same procedure as it does for row 0, except for the obvious difference in the number of bytes that the column is shifted. Listing 14 displays an example for row 1. The value stored at the computed address is loaded with a `lw` instruction, as shown in line 10. After this, the value is shifted and masked (lines 11 and 12) to extract the desired byte at the correct place. The assembly code uses one instruction less for address computation, as explained for row 0. Additionally, the need for a final mask operation is eliminated through the use of a `lbu` instruction instead of a `lw` instruction, which can be seen in Listing 15, line 8.

```
1    ; Pre: a1 contains column 1 of the state.
2    ; Pre: LUT contains the address of the combined look-up table.
3    ; Pre: C contains the value 0xffff.
4    ; Post: t0 contains the obtained S-box value, shifted to the correct place
5    ; in the column.
6    srli    t0, a1, 6
7    andi    t0, t0, 0x1fe
8    slli    t0, t0, 2
9    add     t4, t0, \LUT
10   lw      t0, (t4)
11   slli    t0, t0, 8
12   and     t0, t0, \C
```

Listing 14: computation of `ShiftRows` and `SubBytes` for row 1 (column 0) in the `encrypt_final_round` function of the *generated* assembly code.

```
1    ; Pre: a1 contains column 1 of the state.
2    ; Pre: LUT contains the address of the combined look-up table.
3    ; Post: t0 contains the obtained S-box value, shifted to the correct place
4    ; in the column.
5    srli    t0, a1, 4
6    andi    t0, t0, 0xff0
7    add     t4, t0, \LUT
8    lbu     t0, (t4)
9    slli    t0, t0, 8
```

Listing 15: computation of `ShiftRows` and `SubBytes` for row 1 (column 0) in the `encrypt_final_round` function of the *written* assembly code.

5.1.4 Optimizations and second version

The assembly code utilizes the fact that the right shift to extract a specific byte from the column and the left shift that is needed to compute the table offset in bytes can be combined if the appropriate mask is used. Simply rearranging the order of the operations so that the two shift operations are subsequent to each other in the C code does not help the compiler recognize the possibility of combining the shift operations. However, the table offset can be directly computed in bytes if the `uint32_t` pointer to the encryption table is cast to a `uint8_t` pointer. After

the offset has been added to the look-up table address, the value is cast back to a `uint32_t` pointer to access the look-up table value. Note that the approach for row 0 is slightly different from the other rows because the value needs to be shifted to the left instead of to the right. While this introduces the need for a case distinction within the loop that iterates over the rows, it does not lead to the execution of additional instructions because the loop is unrolled by the compiler.

In the final round, performance can also be improved if the address of the look-up table value is directly computed in bytes. Contrary to the other rounds, the obtained pointer is not cast to a `uint32_t` pointer but is used to extract a single byte from the table. If this approach is employed, the compiler generates a `lbu` instruction instead of a `lw` instruction to load the look-up table value, eliminating the need for an additional mask operation. After the look-up table value has been obtained, it is cast back to a `uint32_t` variable and shifted to place the extracted byte at the desired place in the register.

The dead stores that are present at the end of each round can be eliminated by storing the state in a local array at the beginning of the `encrypt` function and passing the pointer to this local array to the `encrypt_round` function instead of directly passing the `out` parameter, as initially explained in Section 4.2. The inability of the compiler to eliminate the dead stores without this local array can be explained by observing that any changes to the memory location `out` have an impact on the global state of the program. If concurrent behaviour that would access the same memory location is present in the program, eliminating the stores in the function `encrypt` could lead to incorrect results. It should be noted that even though using a local array to store the state improves the performance for the encryption of a single block, this optimization can work conversely for modes of operation like CTR mode, where a local array is already used within the mode to store the counter that is passed to the encryption function.

The improved version (optimization level 1) requires 1189 clock cycles to encrypt a single block of plaintext, which is approximately 6.1% more than the assembly code. Table 3 shows the new distribution of the number of instructions and their types. The implemented optimizations are successful in reducing the number of instructions for the encryption rounds and the final round. The dead stores in the first `AddRoundKey` operation are also eliminated.

5.1.5 Further optimizations and final version

The remaining difference in the performance of the assembly code and the C code can be attributed to the loop overhead, the slightly larger number of callee-saved registers that are used during encryption and a difference in the way that the addresses for the four look-up tables are computed (these are referred to as constants in Table 3). The loop overhead can be eliminated by unrolling the main loop. The implementation with optimization level 2 features this optimization and requires 1149 clock cycles for the encryption of a single block, which is only 2.5% more than the assembly code.

The performance of the C implementation can be improved even further. Each of the three table addresses that are not directly passed to the `encrypt` function is loaded once at the beginning of the function with a `la` (load address) instruction. This is a pseudo instruction and translates to two integer instructions. However, these addresses can be computed with one integer instruction

Instruction purpose	Type	Assembly	C (opt 1)	Total difference
Stack	sw/lw	12	26	+14
	int	2	2	0
Loading / creating constants	int	7	10	+3
First `AddRoundKey`	lw	8	8	0
	int	4	4	0
Loop initialization	int	0	2	+2
Encryption round	lw	20	20	0
(per round)	int	64	64	0
Loop increment and compare	int	0	1	+9
(per round)	branch	0	1	+9
Final round	lw	4	4	0
	lbu	16	16	0
	sw	4	4	0
	int	76	76	0

Table 3: The number and types of instructions for the single-block encryption implementation with optimization level 1 compared to the assembly implementation. In the last column, the number of instructions executed for encrypting one block in the C implementation is compared to the number of instructions executed for the same task in the assembly implementation.

by adding the offsets (four, eight or twelve bytes) to the starting address of the combined look-up table. The code has not been modified to support this optimization because it would not lead to a significant performance gain.

There is no straightforward way to decrease the number of callee-saved registers that are used during encryption in the C code, because this is mainly determined by the register scheduling algorithm of the compiler. However, the overhead that is produced by the additional load and store instructions is small. Moreover, once modes of operation are incorporated into the code, the overhead will be further reduced. The reason for this is that multiple blocks are encrypted within one function call, while stack-related instructions are executed only once.

5.2 Complete AES implementation

Since the performance difference between the C implementation with optimization level 2 and the assembly implementation is within the previously established margin (5%), the performance of the C code is comparable to the performance of the assembly code. Therefore, the AES implementation is completed in C.

5.2.1 Encryption key expansion

The encryption key expansion algorithm has been implemented in C as explained in Section 4.3.1. This implementation is referred to as the encryption key expansion with optimization level 0. Similarly to the single-block encryption implementations, there are two optimized versions of this

algorithm (optimization levels 1 and 2). Table 4 displays a summary of the performance results of the different key expansion implementations.

	Assembly[2]	C (opt 0)	C (opt 1)	C (opt 2)
Clock cycles (absolute)	410	497	443	424
Clock cycles (relative to [Sto19])	0%	+21%	+8.0%	+3.4%

Table 4: Performance in clock cycles of the encryption key expansion process, measured on the Ibex emulator.

Optimization level 0 The algorithmic approach taken in the assembly code is employed to develop the C code with optimization level 0, but a loop is introduced to iterate over the key expansion rounds. Another difference between the two implementations is that the original key is copied into the first four words of the round keys array in the C implementation, whereas this has to be done by the user in the assembly implementation. The C implementation requires 497 clock cycles to complete the key expansion, which is approximately 21% more than the assembly implementation. Table 5 shows the distribution of the number and types of the instructions of the assembly and compiled C code. The four additional `sw` instructions that are needed to store the original key in the round keys array are not considered in the table. The table visualizes that the code for the actual computation of the round keys uses nearly the same quantity and types of instructions in both the compiled C code and the assembly code. One additional `lbu` instruction for each key expansion round is required by the C code to load the round constants. These round constants can be hardcoded in the assembly implementation because the loop that iterates over the key expansion rounds is unrolled.

Instruction purpose	Type	Assembly	C (opt 0)	Total difference
load key	lw	4	7	+3
loading/creating constants	int	4	4	0
loop overhead (once)	int	0	5	+5
loop overhead (per round)	int	0	2	+20
	branch	0	1	+10
round (per round)	int	23	23	0
	sw	4	4	0
	lbu	4	5	+10

Table 5: The number and types of instructions for the encryption key expansion implementation with optimization level 0 compared to the assembly implementation. In the last column, the number of instructions executed for the complete key expansion process in the C implementation is compared to the number of instructions executed for the same task in the assembly implementation.

[2]The C implementation of the key expansion algorithm copies the original key into the first sixteen bytes of the round keys array. This operation is not present in the assembly implementation and would require eight additional clock cycles.

Optimization level 1 The loop overhead accounts for the largest difference in performance, which can be eliminated by unrolling the loop. Note that this would also discard the need for the additional `lbu` instruction. In the implementation with optimization level 1, the main loop has been unrolled. This new implementation requires 443 clock cycles to compute all round keys. A difference of 72 clock cycles is expected (25 for integer instructions, 27 for taken branches and 20 for the `lbu` instructions), but only a difference of 54 clock cycles is found. An analysis of the compiled C code shows that there are nineteen extra `slli` instructions present in the key expansion function. Similar to the `encrypt` function, the compiler does not manage to combine a shift instruction in the C code with the shift instruction that is needed to compute the S-box table offset in bytes, leading to the generation of additional shift instructions. The presence of these instructions in the unrolled version of the loop can again be explained by the compiler's focus on optimizations within loop constructs.

Optimization level 2 The shift instructions are removed by using a different mask in the final key expansion implementation, similar to the approach used in the encryption function. The new implementation (optimization level 2) requires 424 clock cycles to compute the round keys. Eight of the fourteen extra clock cycles, compared to the assembly code, are due to the four additional `sw` instructions needed to store the first round key. The other six clock cycles can be accounted for by the generation of three redundant `lw` instructions at the beginning of the function. It is unclear why these instructions are generated, but this is not further investigated because it only accounts for a 1.5% increase in execution time.

5.2.2 Decryption key expansion

The algorithm of the equivalent decryption key expansion is implemented by generating the encryption round keys with the encryption key expansion function and performing the `InvMixColumns` operation on all round keys except for the first and the last one. The `InvMixColumns` operation is implemented in the same way as in the `encrypt_round` function with optimization level 1/2. One additional application of the S-box is needed to invert the effect of the inverse S-box that is present in the decryption look-up table. This operation is performed in the same manner as in the `encrypt_final_round` function.

The generation of the round keys for the first version (optimization level 0) of the equivalent key expansion algorithm requires 2102 clock cycles. This version uses the encryption key expansion with optimization level 0. An implementation where the loop that iterates over the rounds for the `InvMixColumns` operation is unrolled is also developed (optimization level 2). This implementation uses the encryption key expansion with optimization level 2 and completes the computation process within 2051 clock cycles.

	C (opt 0)	C (opt 2)
Clock cycles (absolute)	2102	2051
Clock cycles (relative to opt 0)	0%	−3.4%

Table 6: Performance in clock cycles of the equivalent decryption key expansion process, measured on the Ibex emulator.

5.2.3 Decryption

The decryption algorithm is implemented at optimization levels 1 and 2. These implementations employ the same approach as the encryption implementations with the same optimization levels. The performance results of the implementations are shown in Table 7. The process of decryption is slightly faster than encryption. In Section 4.3.2, this is said to be expected as a result of the redundancy of mask operations after the inverse S-box lookup in the final round. However, the encryption algorithm at optimization levels 1 and 2 does not use any mask instructions because the value is loaded with a `lbu` instruction. The higher clock cycle count of the encryption algorithm can still be explained. An additional `slli` is needed in the encryption final round for each byte of row 0 to index the look-up table because the S-box elements are stored at each sixteenth byte of the look-up table.

	C (opt 1)	C (opt 2)
Clock cycles (absolute)	1177	1142
Clock cycles (relative to opt 1)	0%	−3.0%

Table 7: Performance in clock cycles of single-block decryption, measured on the Ibex emulator.

5.2.4 CBC mode

A baseline encryption implementation for the CBC mode is developed. The analysis of the generated assembly code reveals the presence of dead stores. Each time the plaintext block is XOR-ed with the previous ciphertext block or with the Initialization Vector, the output is stored at the location of the `out` parameter, even though this memory location is overwritten during the execution of the `encrypt` function. These dead stores are eliminated successfully in the optimized version of the CBC implementation, using the same approach as in Section 5.1.4. The CBC decryption implementation is based on the optimized CBC encryption implementation.

The results of the performance measurements on the Ibex emulator for the encryption and decryption implementations are displayed in Table 8. The results reveal that for inputs of 64 and 256 blocks, the number of clock cycles that is required for decryption is nearly identical to those required for encryption, even though the single-block decryption algorithm is slightly faster than the single-block encryption algorithm. This can be explained by the four extra `lw` instructions that are present in the main loop of the CBC decryption implementation. The presumed purpose of these instructions is to load the previous ciphertext block in registers, which is then XOR-ed with the current output block to obtain the plaintext. In the encryption process, there are no extra `lw`

Number of blocks	4	64	256
CBC encryption baseline	1120.0	1102.2	1101.3
CBC encryption optimized	1111.3	1092.3	1091.3
CBC decryption	1109.5	1092.2	1091.3

Table 8: Average number of clock cycles per encrypted block for CBC mode, measured on the Ibex emulator. The underlying cipher with optimization level 2 is used for all tests.

instructions needed because the produced ciphertext block is immediately utilized in the XOR operation with the next plaintext block.

The test results also show that the number of clock cycles that is required on average to encrypt or decrypt data is larger for small data sizes and stabilizes as the data size grows. This can be accounted for by the overhead that is produced by the function call and the function prologue and epilogue, which becomes neglectable for larger data sizes.

5.2.5 CTR mode

A baseline CTR implementation, which directly follows the specifications of the CTR mode, is developed. Since this implementation uses a local array to store the counter, the optimization to eliminate dead stores described in Section 5.1.4 is not employed. The performance results of this implementation are shown in the first row of Table 9.

To optimize AES in CTR mode, the techniques FACE_{rd0} up to FACE_{rd2} have been implemented. Table 9 displays the results of these optimizations. Concerning the second column of the table, the FACE techniques could only be used for two of the four blocks because the last byte of the initial counter value is set to 255. For this reason, the results for this small input size should not be used to draw any conclusions regarding which FACE techniques are the most beneficial, but are reported nonetheless to show that the performance results somewhat stabilize for larger input sizes. The results show that the isolated use of FACE_{rd0} does not improve the performance, but employing FACE_{rd1} and FACE_{rd2} separately does. While FACE_{rd0} does not reduce the execution time when used in isolation, the performance does benefit from it when it is used in combination with the other FACE optimizations. This observation can be explained by the overhead of checking whether the cache is still valid after the counter value is increased. When FACE_{rd0} is used in isolation, the overhead is too large compared to the benefits of saving three XOR operations in the initial `AddRoundKey` operation. However, when the other FACE optimizations are also employed, this overhead is present anyway, allowing FACE_{rd0} to be used without any additional performance cost. All other combinations of the FACE techniques have been tested, but none of them outperformed the combination of all techniques as shown in the last row of Table 9.

Number of blocks	4	64	256
Baseline	1128	1105	1104
FACE_{rd0}	+26	+15	+14
FACE_{rd1}	−21	−63	−65
FACE_{rd2}	−30	−103	−107
FACE_{rd0} & FACE_{rd1}	−23	−71	−73
FACE_{rd0} up to FACE_{rd2}	−23	−109	−114

Table 9: Average number of clock cycles per block for encryption in CTR mode, measured on the Ibex emulator. The results for the baseline implementation are absolute. All other results are reported relative to this baseline. The underlying cipher with optimization level 2 is used for all tests.

A notable observation is that employing FACE_{rd2} alone leads to a lower clock cycle count than the isolated use of FACE_{rd1}. This is unexpected because three entire columns can be reused in the first round, while only intermediate results can be reused in the second round. Further investigation reveals that whenever FACE_{rd2} can be used for an input block, the compiler generates a shortened version for the first round, essentially employing FACE_{rd1}. This optimization can be performed because, when FACE_{rd2} is used in the second round, only the first column is loaded from the state. The rest of the columns, which are intermediate results from the previously encrypted counter value, are loaded from the cache. Since the values of the last three columns of the state are not accessed, the compiler can remove the computations of these values in the first round.

5.2.6 Expansion to larger key sizes

The complete AES implementation has been extended to support key sizes of 192 and 256 bits in addition to the size of 128 bits. All versions of optimization are available for single-block encryption and decryption, as well as all discussed versions of the modes of operation. The key expansion algorithm for the larger key sizes is only implemented at optimization level 0, but this functionality can also be optimized for these key sizes, using the same techniques that are discussed in Sections 5.2.1 and 5.2.2.

	128		192		256	
	base	opt	base	opt	base	opt
Key expansion encryption	497	−14.7%	511	-	726	-
Key expansion decryption	2102	−2.4%	2514	-	3093	-
Single-block encryption	1384	−17.0%	1650	−18.0%	1896	−17.5%
CBC encryption (average of 64 blocks)	1102	−0.9%	1308	−0.5%	1518	−0.7%
CTR encryption (average of 64 blocks)	1105	−9.9%	1316	−8.5%	1522	−7.3%

Table 10: Performance results for all key sizes, measured on the Ibex emulator. The clock cycle counts of the optimized (opt) implementations are relative to the baseline (base) implementations. The baseline implementations are the implementations that have previously been referred to as ‘baseline‘ or ‘optimization level 0‘. For single-block encryption, the optimized implementation is the implementation with optimization level 2. For the CTR mode, it is the implementation with all FACE techniques enabled.

Table 10 shows the performance of a subset of the AES implementations with all key sizes. Only the performance of the most important implementations is measured and reported because all obtained results align with our expectations. We assume that this is also the case for the other implementations. Regarding the baseline (or optimization level 0) implementations for single-block encryption and the modes of operation, the number of clock cycles grows almost proportionally to the number of rounds performed during encryption, as expected. The table shows that the optimization techniques for single-block encryption and the CBC mode have a similar effect on the larger key sizes as they have on a key size of 128 bits. The FACE techniques that are applied to optimize AES in CTR mode yield a slightly lower performance gain for larger key sizes because the proportion of rounds where FACE can be applied is smaller for these key sizes. This observation aligns with the trend in the paper where the FACE techniques are originally proposed [PL18].

5.3 Verification on the PROACT chip prototype

All tests that are executed on the emulator are also carried out on the PROACT chip prototype. The cryptographic correctness of all implementations is ensured and the execution time is measured in clock cycles. Table 11 shows the obtained performance results for the single-block encryption, decryption and key expansion process. The same information, extended with the performance results on the emulator, is represented visually in Figure 5. All measurements are slightly higher (4% - 12%) than their respective measurements on the Ibex emulator, with most of the measurements being on the low end of this difference. However, it is still the case that each new optimization level for the single-block encryption, decryption, key expansion process and CBC mode offers improvements with regard to the previous level. The performance of the single-block encryption implementation with optimization level 2 is 3.4% slower than the assembly implementation, which is still within the pre-established margin.

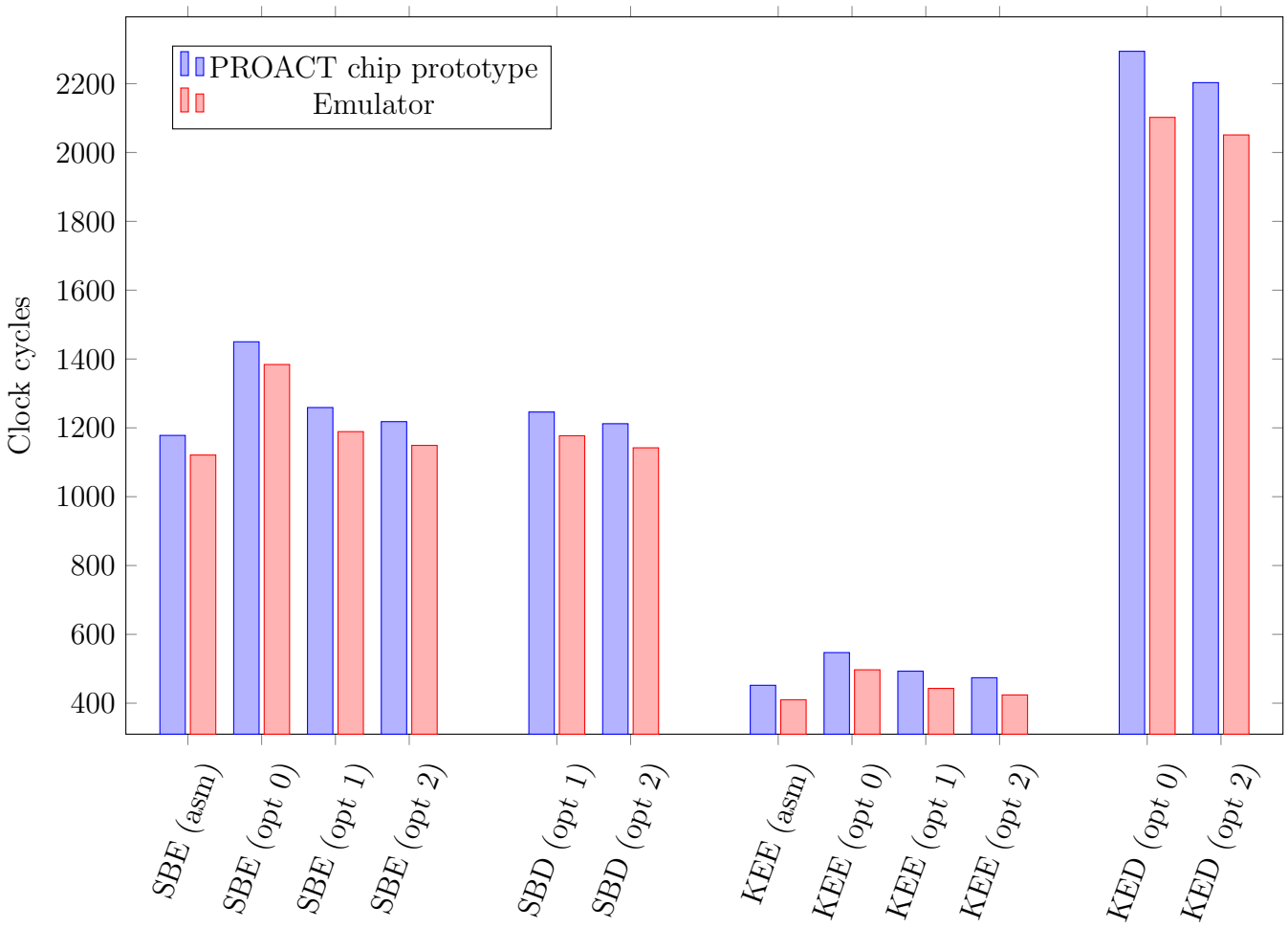

Figure 5: Performance comparison between the PROACT chip prototype and the emulator for the single-block encryption (SBE), single-block decryption (SBD), key expansion encryption (KEE) and key expansion decryption (KED) implementations. All measurements are for the key size of 128 bits.

		Assembly	C (opt 0)	C (opt 1)	C (opt 2)
Single-block encryption	(absolute)	1178	1450	1259	1218
Single-block decryption	(absolute)	-	-	1246	1212
Key expansion encryption	(absolute)	452	547	493	474
Key expansion decryption	(absolute)	-	2294	-	2203
Single-block encryption	(relative to [Sto19])	0%	+23%	+6.9%	+3.4%
Key expansion encryption	(relative to [Sto19])	0%	+21%	+9.1%	+4.9%

Table 11: Performance in clock cycles for a key size of 128 bits, measured on the PROACT chip prototype.

All clock cycle counts for the modes of operation are 4.1%–6.7% higher than their respective measurements on the emulator. The bar plot in Figure 6 displays these differences. The exact clock cycle counts are reported in Table 12. Regarding the CTR mode, it is still the case that employing FACE all through round zero, one and two leads to the best performance for larger inputs. For an input size of four blocks, using only FACE_{rd0} and FACE_{rd1} led to the best result. However, the difference is minor, and as explained in Section 5.2.5, this input size is too small to draw definitive conclusions regarding the most efficient employment of the FACE techniques.

Number of blocks	4	64	256
CBC encryption baseline	1173	1149	1147
CBC encryption optimized	1164	1138	1136
CBC decryption	1162	1139	1127
Baseline	1183	1154	1152
FACE_{rd0}	+37	+20	+19
FACE_{rd1}	−17	−62	−64
FACE_{rd2}	−14	−93	−97
FACE_{rd0} & FACE_{rd1}	−18	−72	−75
FACE_{rd0} up to FACE_{rd2}	−5	−102	−106

Table 12: Average number of clock cycles per block for the modes of operation, measured on the PROACT chip prototype. The results for the optimized implementations in CTR mode are relative to the baseline implementation. The underlying cipher with optimization level 2 is used for all tests.

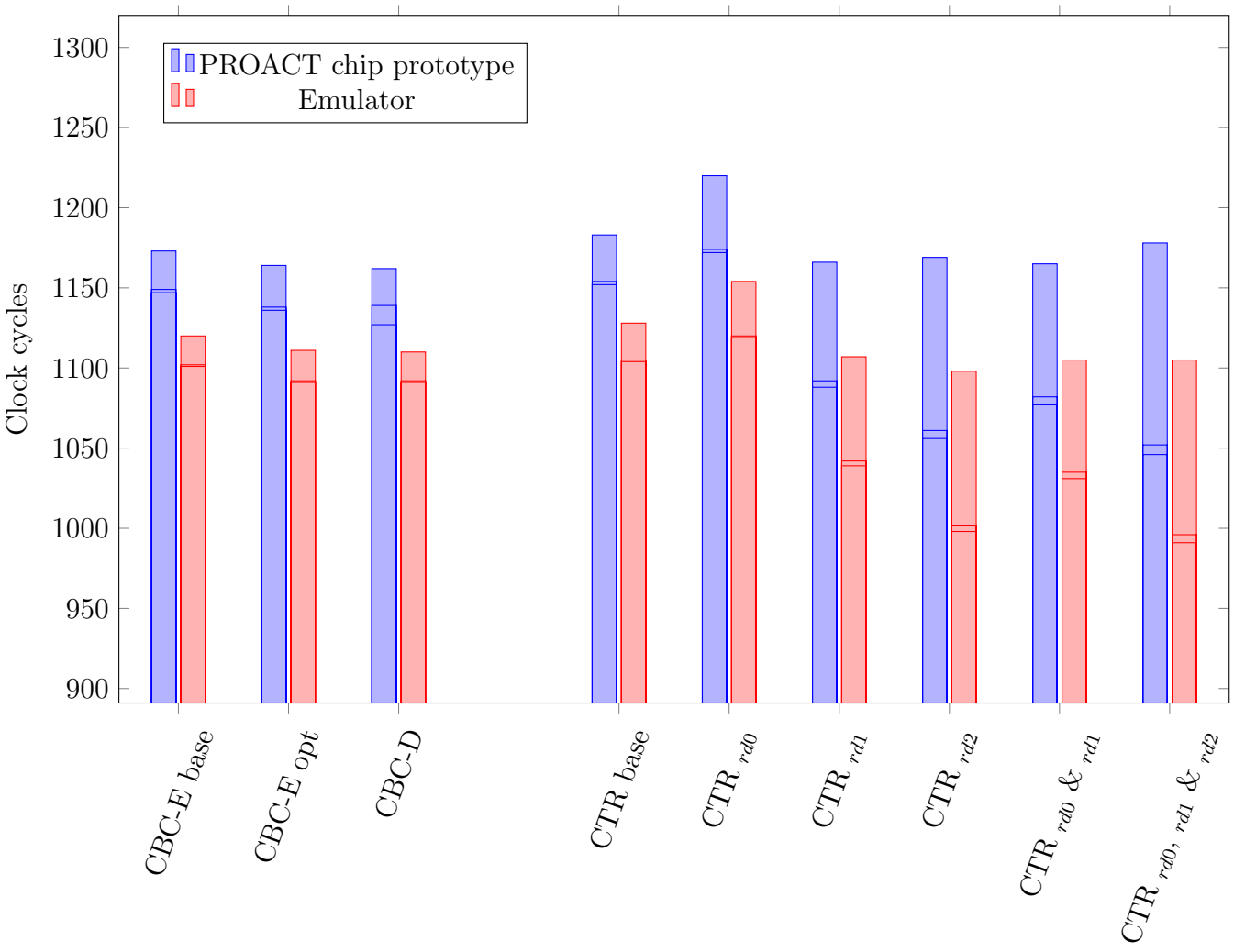

Figure 6: Performance comparison between the PROACT chip prototype and the emulator for the CBC encryption (CBC-E) and CBC decryption (CBC-D) implementations, as well as the CTR mode implementations with different configurations of the FACE techniques. All measurements are for the key size of 128 bits. The full bars represent the average number of clock cycles required for one block, for an input size of 4 blocks. The two lower ticks represent the measurements for input sizes of 64 and 256 blocks, where the input of 256 blocks is represented by the lowest tick out of the two.

6 Discussion

The work, presented in this thesis, aims at developing and optimizing a table-based AES implementation for the PROACT chip, featuring the CBC and CTR mode of operation. The potential for doing this in the C programming language has been explored, revealing that it is possible to develop the encryption algorithm in a manner that allows the compiler to generate code that is only 3.4% slower than the assembly implementation developed by Stoffelen [Sto19]. Two proposed techniques help the compiler recognize the possibility of certain optimizations. First, creating local arrays instead of directly dereferencing the pointers provided in the function parameters reduces the generation of dead stores. Second, for array index computations involving shifts or multiplications, computing the array offset directly in bytes proves beneficial. This can bypass the generations of additional shift instructions in the compiled code. Apart from these techniques, loop unrolling is successfully used to enhance the performance of the implementations.

Only a slight performance improvement has been achieved for the CBC mode of operation by eliminating redundant store instructions. However, implementing three of the five proposed FACE techniques in [PL18] significantly improves the performance of AES in the CTR mode. The combined employment of the three techniques leads to the highest performance.

The clock cycle counts that are measured on the Ibex emulator for single-block encryption, decryption and the key expansion process are compared to measurements obtained from the PROACT chip prototype. While the performance gain that is achieved at different optimization levels is very close to the performance gain on the emulator, the absolute clock cycle counts on the PROACT chip prototype are generally higher than those measured on the emulator.

6.1 Evaluation of key findings

This work reveals that it is possible to develop an efficient AES implementation in a high-level language like C. This is expected, given the advanced optimization techniques available in the GCC compiler used in this research. Another key aspect that contributes to this result is the use of the C programming language as the choice of high-level language. This language is still considerably close to the hardware compared to other high-level languages. For example, it is possible to load four bytes of an array of bytes into one word-sized variable, which can significantly speed up the performance of the AES cipher.

Utilizing local arrays for function parameters and computing array offsets in bytes are techniques found to help the compiler generate more efficient assembly code. The efficiency of the first technique can be explained by the fact that changing a memory location pointed to by a function parameter affects the program's global state, which likely prevents the compiler from performing this optimization automatically. The second technique suggests that the GCC compiler could be improved in its ability to recognize and apply this optimization.

Throughout various parts of the AES implementation, loop unrolling is used as an optimization technique. This is a well-known optimization technique and, in theory, eliminates any loop overhead. However, in the case of the encryption key expansion at optimization level 1, it does not improve

the performance as much as expected. Compilers typically focus their optimizations on loops, as these sections generally consume the most execution time. Thus, when a loop is unrolled, some optimizations that would normally be applied within the loop are not performed.

The paper [PL18] where the FACE optimization techniques are proposed shows a performance gain of 21%–23% for table-based AES with a key size of 128 bits and inputs of 64 and 256 blocks. Yet, a performance gain of only 10% is observed in our work. The results in [PL18] are measured using all five FACE techniques, while our work utilizes three of those techniques. However, the techniques that are not employed can only be applied to inputs larger than 256 blocks, thus they cannot account for the observed difference. One possible explanation for the difference is that the test environments described in [PL18] feature a data cache, allowing the (intermediate) results of previous rounds to be cached. This feature is not present on the PROACT chip. Additionally, the presence of the branch predictor on the test environments in [PL18] could potentially mitigate the overhead associated with checking if the caches require updating. Furthermore, although the Ibex core consistently stalls on a memory load or store, this may not be the case for higher-performance CPUs, thereby reducing the overhead associated with updating and retrieving the results of previous rounds.

The measurements on the PROACT chip prototype consistently show higher cycle counts than those obtained with the emulator. Further investigation reveals that the extra clock cycles are due to a delay in misaligned load and store instructions. When compiling code for the chip, the compiler does not always place byte-sized arrays aligned on a word boundary. This leads to an additional stall cycle during load and store instructions. However, this misaligned placement of byte-sized arrays is not present when the code is compiled for the emulator. Even though the compile options are identical, the linker file that is used for the experiments on the PROACT chip prototype is different from the one that is used for the emulator. This explains the subtle difference in the compiled code.

6.2 Applications and implications

A complete AES implementation optimized for the PROACT chip has been developed. This implementation can be used for efficient data encryption and decryption. Additionally, it can be employed on other RISC-V platforms that do not have a data cache. The provided single-block encryption and decryption functions offer versatility, allowing any mode of operation to be built around it. Furthermore, the development of a graphical user interface that allows a user to conveniently interact with the PROACT chip aids further research and development within the PROACT project.

Apart from the delivered software, the insights that are gained with this work are beneficial for subsequent research. Contrary to some beliefs, efficient AES cryptography does not have to be implemented in assembly. Additionally, the techniques that are applied to help the compiler recognize the possibility of certain optimizations can be applied to optimize other time-sensitive software. Novel compiler optimization techniques that combine subsequent shift instructions in new contexts could also be explored and implemented.

6.3 Limitations

There are several limitations to this work. First of all, while the developed C implementation does have a performance comparable to that of the assembly implementation proposed in [Sto19], it is still slightly slower. If limiting execution time is crucial and code extensibility and portability are not of importance, using an optimized assembly implementation might be preferred.

Secondly, the performance results on the PROACT chip prototype differ slightly from the results on the emulator because of misaligned memory accesses. Further performance improvement on the PROACT chip can be achieved by modifying the memory address of the stack region in the linker file so that this address is divisible by sixteen. Unfortunately, this issue has been identified relatively late in the research process, and there is not sufficient time to test this optimization in the context of this thesis project.

Furthermore, the optimized AES implementation has been tailored specifically to the PROACT chip in its current state. This means that it can not be used for platforms with a data cache, as this would make the implementation vulnerable to cache-based timing attacks [Ber05, BM06]. This vulnerability also implies that the implementation can not be used if the PROACT chip is ever extended to feature a data cache.

Lastly, this work focuses on AES with a key size of 128 bits. While the implementation has been extended to support larger key sizes, there might be optimizations that are beneficial when these key sizes are used that are not present in the current implementation. This is especially applicable to the key expansion algorithm, as it accounts for the most significant differences between the implementations with different key sizes.

6.4 Conclusion

This thesis focuses on developing a table-based AES implementation optimized for the PROACT chip. The implementation is written in the programming language C because it is possible to reach a comparable performance to existing single-block encryption code optimized for the RISC-V ISA. The C implementation offers significant improvements in code readability, portability, and extensibility compared to the assembly implementation, and provides decryption functionalities as well as two modes of operation, that are not present in the assembly implementation. The implementation completes the encryption of one block in 1218 clock cycles on the PROACT chip prototype. To optimize AES in CTR mode, three techniques proposed in the literature are applied, which leads to a performance improvement of approximately 10% compared to the baseline CTR implementation. The optimized implementation can be used for efficient cryptography on the PROACT chip and other RISC-V platforms without a data cache. The graphical user interface that has been developed eases further development and research on the PROACT chip.

References

[ABM04] Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. Efficient AES implementations for ARM based platforms. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, page 841–845, New York, NY, USA, 2004. Association for Computing Machinery.

[AMD24] AMD. Vivado Overview. https://www.xilinx.com/products/design-tools/vivado.html, 2024. Accessed: 25-06-2024.

[AP20] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like Ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021:402–425, 12 2020.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, The University of Illinois at Chicago, 2005.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks Against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 201–215, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[BS08] Daniel J. Bernstein and Peter Schwabe. New AES Software Speed Records. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008*, pages 322–336, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[CJL+20] Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Amber Sprenkels, and Benoit Viguier. Assembly or Optimized C for Lightweight Cryptography on RISC-V? In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security*, pages 526–545, Cham, 2020. Springer International Publishing.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[Dwo01] Morris J. Dworkin. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Technical report, Gaithersburg, MD, USA, 2001.

[ETH20] ETH Zurich and University of Bologna. *Ibex Documentation*, 2020.

[NIS01a] Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. Updated 2023.

[NIS01b] Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

[OBSC10] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast Software AES Encryption. In Seokhie Hong and Tetsu Iwata, editors, *Lecture Notes in Computer Science, vol 6147*, pages 75–93, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[PL18] Jin Park and Dong Lee. FACE: Fast AES CTR mode Encryption Techniques based on the Reuse of Repetitive Data. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 469–499, 08 2018.

[RD20] Mark Randolph and William Diehl. Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman. *Cryptography*, 4(2), 2020.

[RIS24] RISC-V International. *The RISC-V Instruction Set Manual Volume I*, 2024. Available at https://github.com/riscv/riscv-isa-manual/releases/tag/20240411.

[Saj23] Abolfazl Sajadi. PROACT – CARDIS 2023 Poster. https://project-proact.nl/about-the-project/, 2023. Accessed: 18-02-2024.

[Sto19] Ko Stoffelen. Efficient Cryptography on the RISC-V Architecture. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 323–340, Cham, 2019. Springer International Publishing.

[Wu07] HongJun Wu. Hongjun's optimized C-code for AES-128 and AES-256. eSTREAM Project, http://www.ecrypt.eu.org/stream/svn/viewcvs.cgi/ecrypt/trunk/benchmarks/aes-ctr/aes-128/hongjun/v1/?rev=203#dirlist, 2007.

[Xil20] Xilinx. *Python productivity for Zynq (Pynq) Documentation*, release 2.5 edition, Oct 2020. Available at https://pynq.readthedocs.io/en/v2.5/.

A PROACT Chip GUI User Manual

The PROACT Chip GUI is developed to aid further research and development on the PROACT chip. The graphical user interface enables the user to easily execute cryptographic algorithms written in C or RISC-V assembly on the PROACT chip. The GUI application has been designed to be used in a Linux environment in combination with an FPGA development board running the PROACT chip prototype.

A.1 Requirements and setup

The following tools and libraries are required to use the PROACT Chip GUI:

- **riscv32imc GCC toolchain** is needed to cross-compile for RISC-V. The toolchain can be downloaded from https://github.com/lowRISC/lowrisc-toolchains/releases. An example would be 'lowrisc-toolchain-rv32imcb-20240206-1'. The directory in which the toolchain is placed should be added to the `PATH` environment variable and this variable should be accessible in superuser mode.
- **cmake** is needed for the compilation process and can be installed with `sudo apt-get install cmake`.
- **srec_cat** is needed for the compilation process and can be installed with `sudo apt-get install srecord`.
- **xcb-cursor0 or libxcb-cursor0** is needed to start the graphical user interface and can be installed with `sudo apt-get install libxcb-cursor-dev`.

To set up the GUI application, execute the following two steps.

1. Download and extract `PROACT_GUI.tar.xz` from https://git.liacs.nl/s3293912/bachelor_thesis/-/releases.
2. Navigate to the folder that contains `main.py`.
3. The use of a virtual Python environment is recommended to safely access the required packages in superuser mode. Installing these packages directly in superuser mode can lead to broken permissions. To create a virtual Python environment, use the command `python3 -m venv <name_of_venv>`. Then, activate the virtual environment with the command `source <name_of_venv>/bin/activate`.
4. Install all necessary Python packages inside the virtual environment by running the command `pip install -r python_requirements.txt`.

A.2 Usage

To use the GUI, connect the FPGA board to your computer and upload the bitstream file `PROACT_top.bit` to the board. This can, for instance, be done with Vivado [AMD24]. The bitstream file can be found in the downloaded folder. Then, navigate to the folder that contains `main.py`

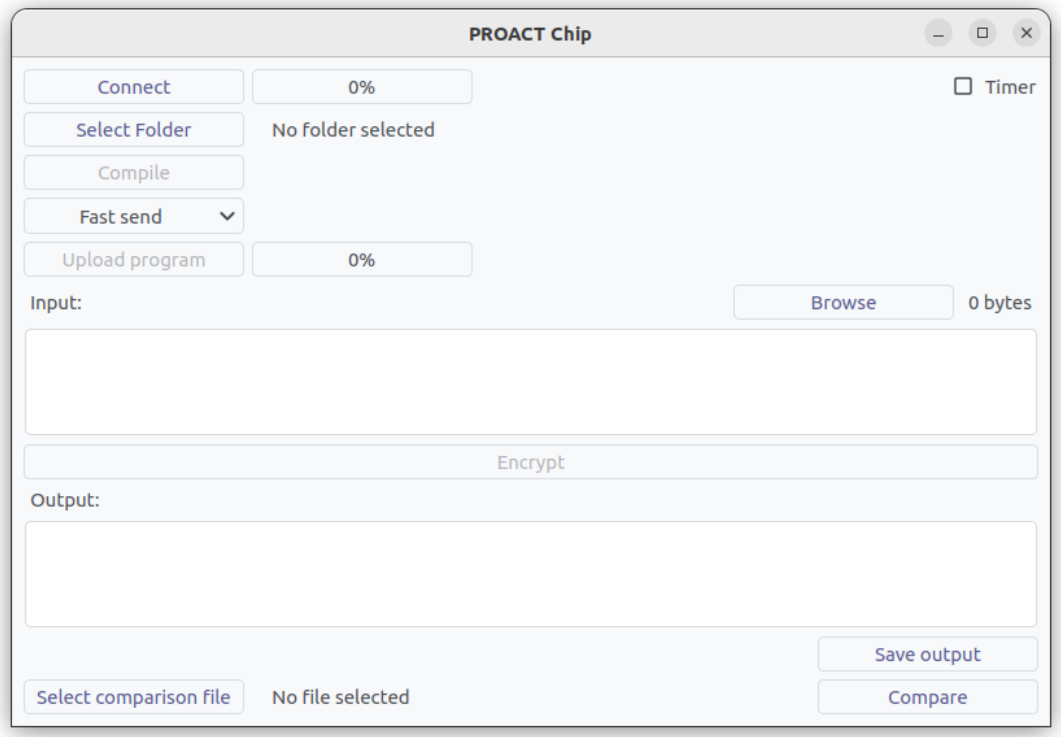

Figure A.1: The PROACT Chip graphical user interface

and start the GUI with the command `sudo su -c "<name_of_venv>/bin/python3 main.py"`. It is necessary to run the GUI in superuser mode to communicate with the board and access peripherals.

Figure A.1 shows the application upon startup.

- The **connect** button initiates the connection between the computer and the FPGA board.
- The **select folder** button is used to choose a folder that contains the source code for an experiment.
- The **compile** button compiles the code from the selected folder and converts it to a VMEM file.
- The **fast send/secure send** drop-down list is used to select a send method. In most cases, 'fast send' is preferred. 'Secure send' verifies whether the data is correctly written to the memory. This can be useful for debugging purposes.
- The **upload program** button uploads the generated VMEM file to the memory of the chip.
- The **input** field can be used to provide a hexadecimal input for the encryption algorithm. Alternatively, the **browse** functionality can be used to select an input file. The input field can also be left empty.
- The **encrypt** button starts the execution of the uploaded program. Once the execution is finished, the output of the algorithm is displayed in the **output** field. The output has the same length as the provided input.

- The **save output** button can be used to save the output to a text file.
- The **select comparison file** and **compare** buttons can be used to compare the algorithm output with a text file of choice.
- The **timer** functionality can be enabled to measure the execution time of the algorithm. See Section A.3 for instructions on how to use this functionality. The timer option is disabled by default and should only be enabled if the uploaded program uses the timer.

A.3 Notes on developing experiments for the PROACT Chip GUI

Code structure To ensure that the user-provided program can be compiled correctly, a one-level directory structure should be used. The use of subfolders within the main program folder is currently not supported due to time constraints, but this functionality could be implemented in a future version of the GUI application. Additionally, the main function should be written in C and should always contain a return statement.

Input and output When programming in C, the input and output of the algorithm can be accessed using the pointers `uint8_t* PROACT_IN` and `uint8_t* PROACT_OUT`, respectively. The pointers need to be declared by the user before use, as shown in Listing A.1. Alternatively, two arrays with the same names as the pointers can be declared, e.g. `uint8_t PROACT_IN[16]`. This method is shown in Listing A.2. During the compilation process, code will be generated to make these pointers reference the memory locations `0x800F400` (input) and `0x800F800` (output). When programming in assembly, these addresses can directly be accessed to control the input and the output. The size of the input and the output arrays is limited to 1024 bytes each.

Timer The timer functionality can be used to measure the execution time of an algorithm. The user can specify the moments to start and stop the timer by making the following function calls: `Trigger(1)` (start the timer) and `Trigger(0)` (stop the timer). Listing A.2 shows an example where the execution time of the for-loop is measured. When these function calls are inserted in the source code and the timer function in the GUI is enabled, a message will appear once execution has finished reporting the measured time in clock cycles and nanoseconds for a clock frequency of 50 MHz.

```
1  int main() {
2      uint8_t* PROACT_IN;
3      uint8_t* PROACT_OUT;
4
5      // copy input to output
6      for (int i = 0; i < 16; i++)
7          PROACT_OUT[i] = PROACT_IN[i];
8
9      return 0;
10 }
```

Listing A.1: Using pointers to access the input and output.

```
int main() {
    uint8_t PROACT_IN[16];
    uint8_t PROACT_OUT[16];

    Trigger(1);

    // copy input to output
    for (int i = 0; i < 16; i++)
        PROACT_OUT[i] = PROACT_IN[i];

    Trigger(0);

    return 0;
}
```

Listing A.2: Using arrays to access the input and output and using the timer functionality.

B AES Implementation Code

The Git repository containing all AES implementations that are developed for this thesis project is available at https://git.liacs.nl/s3293912/bachelor_thesis.

B.1 Code repository structure

The repository is structured into three folders.

- The folder `GUI` contains the source code for the graphical user interface. Appendix A provides a manual for the GUI.
- The folder `aes` contains the developed AES implementations, as well as the source code for the experiments that have been carried out on the Ibex emulator with Simple System.
- The folder `aes_on_PROACT` contains the developed AES implementations, as well as the source code for the experiments that have been carried out on the PROACT chip prototype. Note that the files regarding the AES implementations (`aes.c`, `aes.h` and `tables.h`) are identical in the folders `aes` and `aes_on_PROACT`. The files are present in both folders to ease testing on the PROACT chip prototype.

B.2 Instructions to run experiments

The code repository provides files to run experiments on the Ibex emulator, as well as on the PROACT chip prototype. As an example, let us consider the implementation for single-block encryption at optimization level 0 which is described in Section 5.1.1. The experiment can be configured in `aes/experiment_config.h` (for the Ibex emulator) or `aes_on_PROACT/experiment_config.h` (for the PROACT chip prototype). Both files define the same parameters, which are explained in Table B.1. To configure the implementation of Section 5.1.1, the settings in the last column should be used. Note that the different modes of operation, including single-block encryption, can be enabled simultaneously, but this causes the clock cycle count to accumulate. Therefore, such a configuration can not be used to obtain performance results. The same concept applies to the parameters that disable and enable encryption and decryption.

In the file `experiment.c`, which is present in the folder `aes` as well as `aes_on_PROACT`, the function calls to enable and disable the timer are placed so that the clock cycles for the encryption or decryption process are measured. If measuring the performance of the key expansion process is desired, these function calls need to be moved so that they are placed around the function call to the key expansion function. Information on the timer function for the emulator (`pcount_enable`) is provided in Section 4.5 and Appendix A.2 provides information on the timer function for the PROACT chip (`Trigger`).

The code for the experiment is now correctly configured. The remainder of this section is split into two parts because the execution of the experiment on the emulator differs from the execution on the PROACT chip prototype.

Parameter	Explanation	Possible values	Example
KEYSIZE	Size of the encryption key (in bits)	128, 192, 256	128
OPT_KS	Optimization level of the key expansion	0, 1, 2 [3]	2
OPT_CIPHER	Optimization level of the AES cipher	0, 1, 2	0
SINGLE_BLOCK	Dis- or enable single-block encryption	0, 1	1
CBC	Dis- or enable the CBC mode	0, 1	0
CTR	Dis- or enable the CTR mode	0, 1	0
OPT_CBC	Dis- or enable optimization for CBC mode	0, 1	x
CTR_CACHE_0	Dis- or enable FACE_{rd0} for CTR mode	0, 1	x
CTR_CACHE_1	Dis- or enable FACE_{rd1} for CTR mode	0, 1	x
CTR_CACHE_2	Dis- or enable FACE_{rd2} for CTR mode	0, 1	x
ENCRYPTION	Dis- or enable encryption	0, 1	1
DECRYPTION	Dis- or enable decryption	0, 1	0
INPUT_SIZE	Size of input for CBC or CTR mode (in bytes)	64, 1024, 4096	x

Table B.1: Possible experiment parameters. The last column shows the configuration for the single-block encryption implementation with optimization level 0, as described in Section 5.1.1. An 'x' indicates that the parameter does not have any influence on the outcome of the experiment.

Ibex emulator Simple System is required to run the experiments inside the `aes` folder. For instructions on how to set up and use Simple System, refer to https://github.com/lowRISC/ibex/tree/master/examples/simple_system. The folder `aes` contains a makefile that can be used in combination with Simple System. In the last line of this file, the path to the Ibex repository containing Simple System must be specified. An example of the bash commands that can be used to run the experiment after configuring the makefile is shown in Listing B.3. The paths can differ based on the Simple System installation.

PROACT chip prototype The PROACT Chip GUI is required to run the experiments inside the `aes_on_PROACT` folder. For instructions on how to set up and use the GUI, refer to Appendix A. Once the GUI is set up, select the `aes_on_PROACT` folder in the GUI and compile and upload the experiment. To run the experiment, provide an input of choice, e.g. `aes_on_PROACT/testfiles/AES128_16_in1.txt`, and select the 'encrypt' button.

```
1 cd <path_to>/ibex
2 make clean -C <path_to>/bachelor_thesis/aes
3 make -C <path_to>/bachelor_thesis/aes
4 ./build/lowrisc_ibex_ibex_simple_system_0/sim-verilator/Vibex_simple_system \
5     [-t] --meminit=ram,<path_to>/bachelor_thesis/aes/experiment.elf
```

Listing B.3: Example of bash commands to run an experiment with Simple System

[3] The key expansion optimization levels 1 and 2 are only available for a key size of 128 bits.