Rijndael Encryption implementation on different platforms, with emphasis on performance

KAFUUMA JOHN SSENYONJO

Bsc (Hons) Computer Software Theory

University of Bath

May 2005

Rijndael Encryption implementation on different platforms, with emphasis on performance.

Submitted by Kafuuma John Ssenyonjo

Acknowledgements

Firstly I would like to thank my project supervisor Dr. Russell Bradford for his help and ideas. I would also like to thank my personal tutor Dr. A.M. Barry for his suggestions. Finally I wish to thank my parents for their ongoing support, my brother Roger Mbuga for proof reading and the rest of my family and friends who have always been and remain supportive of me.

Abstract

This dissertation implements the Advanced Encryption standard (Rijndael) and compares how it performs on different platforms. Rijndael is a key-iterated block cipher with a fixed block length of 128 bits, and supports key lengths of 128, 192 and 256 bits. In this dissertation the Advanced Encryption algorithm has been implemented in C++ and 68000 assembly languages. The performance figures for the Rijndael 68000 assembly programs were obtained by calculating using the instruction execution times of the 68000 microprocessor, while those for the Rijndael C++ implementation were obtained by performance testing the implementation on the following processors: Intel Celeron, Intel Xeon and Ultra SPARC processors. The dissertation is concluded with a discussion of the performance figures obtained from performance testing the variations of Rijndael implementations on the different platforms and how they compare with the performance figures that are already published. Also discussed are the further improvements that could have been implemented to enhance the performance of the implementations.

Contents

Chapter 1: Introduction

1.1 Introduction

The Advanced Encryption Algorithm (Rijndael) is a block cipher that was designed to replace the now old Data Encryption Standard (DES). According to the "The Design of Rijndael" [1], a *block cipher* transforms *plaintext blocks* of a fixed length *n* to *ciphertext blocks* of the same length under the influence of a cipher key *k*. *Encryption* is the operation of transforming a plaintext block into a ciphertext block and the operation of transforming a cipher text block into a plaintext block is called *decryption* [1].

1.2 Data Encryption Standard

The Data Encryption Standard (DES) is a widely used algorithm for encrypting data and was adopted as a standard in January of 1977. The DES algorithm was designed to encipher and decipher blocks of data consisting of 64 bits under control of a 56 bit key (64 bit key including parity), by means of permutation and substitution. The Data Encryption standard takes a 64 bit (8 byte) block of plaintext and subjects it to 16 iterations or "rounds" of a complex function to produce 64 bits of ciphertext, Beckett [2].

Since the DES key length is only 56 bits, DES can be, and has been broken by brute force attack methods of running through all possible keys. At the time the key was chosen in the mid 1970s', computers in the mainstream did not posses enough power to brute force break keys of this length. With computer hardware progress in years since, almost anyone can have the sufficient computational capacity to break the DES key by brute force. Both differential and linear were also successfully applied to Data Encryption Standard: *differential cryptanalysis* was the first chosen plaintext attack, and *linear cryptanalysis* was the first known-plaintext attack that is theoretically more efficient than an exhaustive key search for the DES [1].

On 2 October, 2000, the US National Institute of Standards and Technology (NIST) officially announced that, Rijndael would become the Advanced Encryption Standard (AES). The Rijndael algorithm was designed by two well-known specialists, Joan Daemen and Vincent Rijmen from Belgium. The new encryption standard was to become a Federal Information Processing Standard (FIPS), replacing the old DES and triple-DES. The AES is used to protect sensitive information of several government organisations, as well as private businesses.

1.3 Aim

The overall aim of this dissertation is to measure the performance of the variations of the AES (Rijndael) on different platforms, and based on the results recommend which variation is best suited for a particular platform. These platforms being 8 bit, 32 bit, 64 bit and multi-processor platforms.

There has been work done on the performance of Rijndael on the 8 bit and 32 bit platforms. On the 8 bit processors, Rijndael has been implemented in assembly language for two microprocessors that are representative for Smart Cards (credit/ debit cards) in use today. These are Intel 8051 and Motorola 68HC08 microprocessors, and the performance figures are included in "The Rijndael Block Cipher" [3]. There are also published performance figures by Brian Gladman [3] and [4], on 32-bit processors, where Rijndael was implemented in C/C++ and compiled in Microsoft Visual C++ (version 6) on Pentium Pro and Pentium II processors.

The aim of this dissertation will achieve some of what has already been done on 8 bit and 32 bit platforms. This will justify the efficiency (or otherwise) of the software implementation of the Rijndael algorithm on these platforms.

1.3.1 Objectives

- Implement the Rijndael algorithm in a preferred high-level language.
- Implement the Rijndael algorithm in a preferred assembly language.
- Compile and performance test the implemented assembly language source code on an 8 bit platform.
- Compile and performance test the implemented high-level language source code on 32 bit platform.
- Compile and performance test the implemented high-level language source code on 64 bit platform.
- Compile and performance test the implemented high-level language on multi-processor platform.

1.4 Report Structure

The rest of this dissertation is structured as follows. Chapter 2 is the literature review which provides background information that is used throughout this document. Chapter 3 provides the requirement specification and analysis for this dissertation. Chapter 4 and 5 discuss in detail how the Rijndael algorithm was implemented for this project and why particular methods of implementation were chosen over others. In Chapter 6 the testing and performance of the AES implementations is discussed by analysing the performance results. Finally Chapter 7 offers a critical evaluation of the dissertation, highlighting the achievements and suggesting future improvements.

Chapter 2: Literature Review

2.1 Mathematical Background

The following mathematical concepts are useful in order to understand the Rijndael encryption algorithm.

2.1.1 Algebraic Properties

The set of all integers denoted by $\mathbb{Z}$ is closed under the operations of subtraction, addition and multiplication. $\mathbb{Z}$ is not closed under the division operation since the quotient of two integers (e.g. 1 divide by 2) is not always an integer.

The following is a lists some of the basic properties of multiplication and addition for any integers a, b and c.

Closed: $(a + b)$ or $(a \times b)$ is an integer.

Associative: $(a + (b + c)) = ((a + b) + c)$ or $(a \times (b \times c)) = ((a \times b) \times c)$

Commutative: $(a + b) = (b + a)$ or $(a \times b) = (b \times a)$

Existence of *neutral element*: $(a+ 0) = a$ or $(a \times 1) = a$, neutral elements have no effect on other elements when combined with them.

Existence of *inverse element*: $(a + -a) = 0$, inverse of an element reverses the effect that element.

Distributive: $a \times (b + c) = (a \times b) + (a \times c)$

In abstract algebra, a *field* is an algebraic structure in which the operations of addition, subtraction, multiplication, and division (except division by zero) may be performed and the associative, commutative, and distributive rules hold, which are familiar from the arithmetic of ordinary numbers [5]. The set of all integers $\mathbb{Z}$, has no multiplicative inverse $(a \times a^{-1} = 1)$ because it is not closed under the division operation, therefore $\mathbb{Z}$ not a field.

2.1.2 Finite Field

A *finite field* (Galois field) is a field with finite number of elements and has a prime characteristic. The number of elements in a set is called the *order* of the field and fields of the same order are called *isomorphic*, since they have a similar structure. According to [1], Rijndael uses finite fields that have characteristic 2.

Elements of a finite field $GF(p)$ can be represented by the integers 0 to $[p-1]$, where p is the characteristic of field. Rijndael algorithm uses finite fields with characteristic 2, therefore there only two elements $\{0,1\}$.

2.1.3 Euclidean algorithm

The *Euclidean algorithm* determines the greatest common divisor (gcd) of two integers. The greatest common divisor of two integers is the largest number that divides both integers, if both integers are not zero. For example gcd of 10 and 75 is 5.

The *extended Euclidean algorithm* is a version of the Euclidean algorithm; its input are two integers a and b then the algorithm computes their greatest common divisor (gcd) as well as integers x and y such that $ax+by=\gcd(a,b)$. This works because the steps of Euclid's algorithm always deal with sums of multiples of a and b .

The equation $ax+by=\gcd(a,b)$, is particularly useful when a and b are coprime (meaning a and b have gcd 1): x is then the multiplicative inverse (reciprocal) of a modulo b .

In modular arithmetic, the multiplicative inverse of x can also defined: it is the number a such that $(a\times x)\bmod n=1$. However, the multiplicative inverse only exists if, a and n are coprime. For example, the inverse of 3 modulo 11 is 4 because it is the solution to $(3\times x)\bmod 11=1$ .The extended Euclidean algorithm may be used to compute the multiplicative inverse modulo of a number [5].

2.1.4 Finite field $GF(2^8)$ arithmetic

Elements of a finite field can be expressed in various numerical forms, in Rijndael algorithm specification the polynomial representation was chosen, with each term representing a bit in the elements' binary expression.

Example 2.1: the hexadecimal value $\{4\mathrm{a}\}$ which is $\{01001010\}$ in binary will be represented as the following polynomial x^6+x^3+x.

Addition

In a finite field of characteristic 2 as in Rijndael, the sum and difference of two elements are identical operations and equivalent to a simple bitwise exclusive OR between the terms of both elements.

Example 2.2: Given two hexadecimal values $\{34\}$ and $\{4e\}$ their finite field sum or difference which is an exclusive OR (denoted by $\oplus$) is,

Hexadecimal: $\{34\} \oplus \{4e\} = \{3a\}$
Binary: $\{01110100\} \oplus \{01001110\} = \{00111010\}$
Polynomial: $\left(x^6 + x^5 + x^4 + x^2\right) \oplus \left(x^6 + x^3 + x^2 + x\right) = \left(x^5 + x^4 + x^3 + x\right)$

Multiplication

Multiplication in the finite field $GF\left(2^8\right)$ is multiplication followed by division using the irreducible polynomial used to define the finite field as the divisor, and the remainder is the product. A polynomial is irreducible if it has no other divisors other than 1 and itself. For Rijndael algorithm the irreducible polynomial below denoted $m(x)$ is used.
$m(x) = x^8 + x^4 + x^3 + 1$

Example 2.3: $\{34\} \bullet \{4e\} = \{f0\}$ where the symbol "•" is used to denote multiplication in a finite field.

$\left(x^6 + x^5 + x^4 + x^2\right) \bullet \left(x^6 + x^5 + x^4 + x^2\right) =$

$\left(x^{12} + x^{11} + x^{10} + x^8\right) \oplus \left(x^{11} + x^{10} + x^9 + x^7\right) \oplus \left(x^{10} + x^9 + x^8 + x^6\right) \oplus \left(x^8 + x^7 + x^6 + x^4\right) =$

The intermediate result is the finite field sum $= x^{12} + x^{10} + x^8 + x^4$
which is then divided by $m(x)$ above;

$\left(x^8 + x^4 + x^3 + 1\right) \times x^4 \quad = \quad x^{12} + x^8 + x^7 + x^4$

Subtract to get intermediate reminder $\quad = \quad x^{10} + x^7$

$\left(x^8 + x^4 + x^3 + 1\right) \times x^2 \quad = \quad x^{10} + x^6 + x^5 + x^4$

Subtract to get final reminder $\quad = \quad x^7 + x^6 + x^5 + x^4$

Therefore $\left(x^6 + x^5 + x^4 + x^2\right) \bullet \left(x^6 + x^5 + x^4 + x^2\right)$ modulus $m(x) = x^7 + x^6 + x^5 + x^4$, and is the product of finite field multiplication.

Multiplication by Repeated Shifts

Finite field multiplication can also be achieved using finite field element $\{00000010\}$ the polynomial x .Given that when another polynomial element is multiplied by polynomial x (hexadecimal value $\{02\}$) all its powers of x incremented by 1. This is equivalent to shifting its byte representation by one bit so that a bit at position i moves to position $i+1$. If the top bit is set prior to this move it will overflow to create an x^8 term, in which case the result is exclusive OR-ed with modular polynomial $m(x) = x^8 + x^4 + x^3 + 1$, leaving a result that fits within a single byte. Therefore by repeating this process, a finite field element can be multiplied by all powers of x form 0 to7. The finite field product of this element and any other finite field element can then be achieved by exclusive OR-ing the results for the appropriate powers of x. The example below taken from the Rijndael specification [6], illustrates multiplication by repeated shifts.

Example 2.4: The finite field multiplication of two finite field elements $\{57\}$ and $\{83\}$ using repeated shifts to give $\{c1\}$.

P	**{57}** $\bullet$ x^p	**XOR** $m(x)$	$\{57\} \bullet x^p$	$\{83\}$	**XOR value**	**result**
0	{01010111}		{01010111}	1	{01010111}	{01010111}
1	{10101110}		{10101110}	1	{10101110}	{11111001}
2	1{01011100}	1{00011011}	{01000111}	0	{00000000}	{11111001}
3	{10001110}		{10001110}	0	{00000000}	{11111001}
4	1{00011100}	1{00011011}	{00000111}	0	{00000000}	{11111001}
5	{00001110}		{00001110}	0	{00000000}	{11111001}
6	{00011100}		{00011100}	0	{00000000}	{11111001}
7	{00111000}		{00111000}	1	{00111000}	{11000001}

Figure 2.1: Finite field multiplication by repeated shifts

Finally as described in the specification for the Rijndael algorithm [6], finite field multiplication can also be accomplished by using two 256 byte logarithm tables.

2.1.5 Polynomials with coefficients in $GF(2^8)$

Polynomials of degree less than 4 that are representative of a 32 bit word in the internal state of the Rijndael can be defined with the coefficients that are finite field elements. An example is, $(b_3x^3 + b_2x^2 + b_1x + b_0)$, where the coefficients (b_3, b_2, b_1, b_0) each represent a byte in the 32-bit word.

Finite field addition for these polynomials is achieved by adding the coefficients of like power of x., which is equivalent to exclusive OR-ing the corresponding bytes.

As described in the FIPS standard197 [7], multiplication is achieved by polynomial product and reducing the product modulo a polynomial of degree 4, so that the product is a polynomial of degree less than 4. For Rijndael algorithm, the polynomial $x^4 + 1$ is used as the 4 degree polynomial.

2.2 Algorithm Specification

Rijndael is a key-iterated block cipher with both a variable block length and a variable key length. That is both the key size and the block size may be chosen to be any of 128, 192, or 256 bits. The only difference between Rijndael and the AES is the range of supported values for the block length and cipher key length. The AES fixes the block length to 128 bits, and supports key lengths of 128, 192 or 256 bits only. The extra block and key lengths in Rijndael were not evaluated in the AES selection process, and are not adopted in the current FIPS standard.

2.2.1 Rijndael Byte and State

The input and output for the AES (Rijndael) is a sequence of 128 bits, which is the size of the cipher block cipher and the cipher key length can be 128, 192 or 256 bits. A Byte in Rijndael is a collection of 8 bit sequences and represents a finite field element. Using the polynomial representation of finite field elements, the byte *b* is represented as follows:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

The coefficients $\left(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\right)$ are either 1 or 0 given that Rijndael uses the finite field $GF\left(2^8\right)$. Byte values in Rijndael are represented using hexadecimal notation, with each of the two of four bits being denoted by a character. For example binary $\{10001010\}$ is represented as hexadecimal $\{8a\}$ and is polynomial $x^7 + x^3 + x$.

The state in Rijndael as defined in the specification [6] is a two dimensional array of bytes, that contains 4 rows and N_c, where N_c is the input sequence length divided by 32. For the AES N_c is 4 as cipher block size is fixed to 128. Each column in the state is a four byte array and therefore can be thought of as a 32-bit word.

2.2.2 The Rounds

Rijndael has a variable number of *rounds* that depend on the cipher block length and the key length as shown below:

- 10 if both the block and the key are 128 bits long.
- 12 if either the block or the key is 192 bits long, and neither of them is longer than that.
- 14 if either the block or the key is bits long.

An encryption with Rijndael consists of an initial key addition, the *AddRoundKeys* step followed by the regular *rounds*, noted above, but with the final *round* omitting the *MixColumns* step. In the initial *AddRoundKeys* step the block cipher is exclusive OR-ed with a round key (described in section 2.4).

Each regular *round* is a sequence of four byte-oriented transformations, called *steps*. When using the Rijndael algorithm for encryption these *steps* are performed in the following order; *SubBytes*, *ShiftRows*, *MixColumns* and finally *AddRoundKeys* step.

2.2.3 SubBytes Step

In the SubBytes transformation each byte the state is replaced by a new byte value that is derived from the S-box substitution table as shown in the Rijndael specification [6]. This byte valves in the substitution table are constructed by composing two transformations:

- First replace the each state byte with its reciprocal (multiplicative inverse) in the finite field $GF\left(2^8\right)$. Zero has no reciprocal so it is replaced by its self.
- Secondly the reciprocal bytes are transformed using an affine transformation over $GF(2)$. This involves a finite field multiplication by a matrix M, followed by finite field addition (exclusive OR) to a vector V of hexadecimal value $\{63\}$ .Given a byte b , the affine transformation is equal to $\{b\}\times[M]+\{V\}$,as shown below in detail;

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 10001111 \\ 11000111 \\ 11100011 \\ 11110001 \\ 11111000 \\ 01111100 \\ 00111110 \\ 00011111 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

2.2.4 ShiftRows Step

The ShiftRows transforms the state by cyclically shifting the bytes in the last three rows as follows for the AES block:

From ***To***

$$\begin{bmatrix} B_1, B_5, B_{9,} B_{13} \\ B_2, B_6, B_{10,} B_{14} \\ B_3, B_7, B_{11}, B_{15} \\ B_4, B_8, B_{12}, B_{16} \end{bmatrix} \qquad \begin{bmatrix} B_1, B_5, B_{9,} B_{13} \\ B_6, B_{10,} B_{14}, B_2 \\ B_{11}, B_{15}, B_3, B_7 \\ B_{16}, B_4, B_8, B_{12} \end{bmatrix}$$

The shift amount depends on the row number and the block length as shown in the table in Rijndael specification [6]. For the AES block above the last three rows are shifted 1, 2 and 3 places to the left respectively.

2.2.5 MixColumns Step

The MixColumns transformation treats each column of the state as four term polynomial over $GF\left(2^8\right)$ as described in section 2.1.5 and multiplied modulo $x^4 + 1$ by a fixed polynomial $a\left(x\right)=\left\{03\right\}x^3+\left\{01\right\}x^2+\left\{01\right\}x+\left\{02\right\}$. This is actually a matrix multiplication (finite field multiplication) of the each column with the matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

2.2.6 AddRoundKeys Step

The AddRoundKeys transformation is the final step and simply performs a finite field addition (exclusive OR) between the state columns and the round key (described in section 2.4) of the current round.

2.3 The Inverse Cipher

The Rijndael algorithm for decryption is achieved by using the inverses of the *SubBytes*, *ShiftRows, MixColumns* and *AddRoundKeys* transformations described earlier in the chapter, with their order reversed and also using the same key schedule as the cipher in reverse. The AddRoundKeys step is its own inverse and is therefore inverse *AddRoundKeys* is also an exclusive OR operation. The inverse *MixColumns* transformation uses the inverse of the fixed matrix (shown in section 2.2.5) used in the cipher *MixColumns* step. The inverse matrix used is shown below:

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix}$$

In the inverse *SubBytes* step the inverse of the S-box is applied to each byte of the state and in the inverse *ShiftRows* step the last three rows are shifted 1, 2 and 3 places to the right respectively.

The order of some of the operations of the inverse cipher can be changed without affecting the final result. The inverse *SubBytes* step could just as easily be done after the inverse *ShiftRows* step before it because *SubBytes* transformation changes the values of the bytes without changing their positions whereas *ShiftRows* transformation does the exact opposite.

According to the FIPS standard [7], the order of the inverse *AddRoundKeys* and inverse *MixColumns* operations can be inverted to put the cipher and inverse cipher in the same form provided that an adjustment is made to the key schedule. This is because of the column mixing operation is linear with respect to the column input so that:

$$inverseMixColumns(state \oplus roundkey) = inverseMixColumns(state) \oplus inverseMixColumns(roundkey)$$

Where $\oplus$ is an exclusive OR. This operation is not performed on the first and last round keys as the inverse *MixColumns* is not performed on them. By transforming the inverse cipher into the same sequence as the cipher, it can be expressed in an efficient form for implementation.

2.4 Key Schedule

The round keys are derived from the cipher key by means of a key schedule. The AES algorithm takes the cipher key and performs a key expansion to generate a total of $N_b(N_r+1)$ 32-bit words. N_b is the number of column in the state which is 4 for AES and N_r is the number of rounds which is = 10, 12 or 14 for AES.

Each of the rounds in Rijndael including the initial *AddRoundKeys* step requires a set of four 32-bit words of key data. The key expansion algorithm expands the input cipher key by filling the first N_k words with the input cipher key. N_k is number of 32-bit words comprising the cipher key, which for AES is = 4, 6 or 8. Then every following word $w[i]$ is equal, the exclusive OR of the previous word $w[i-1]$ and the word N_k words earlier $w[i-N_k]$. For the words in positions that are a multiple of N_k, the previous word $w[i-N_k]$ is first rotated one byte to the left and then its bytes are transformed using the S-box from the *SubBytes* step and then is exclusive OR-ed with a round-dependent constant prior to the exclusive OR with $w[i-N_k]$.

The key expansion for the input cipher key of bit length of 256 bits $(N_k=8)$ differs slightly from that of 128 and 192 bit lengths. In addition to the above, when $N_k=8$ and byte in position $[i-4]$ is a multiple of 8, then the previous byte is transformed using the S-box from the *SubBytes* step prior to the exclusive OR with $w[i-N_k]$. In the appendix of the FIPS [7], there are examples of key expansions of the various cipher keys of the AES.

2.5 Algorithm Implementation

The Advanced Encryption standard (AES) can efficiently be implemented in software and dedicated hardware. Given that the Rijndael implementation in software on general purpose processors is already very fast, the hardware implementation is needed in few cases.

On the 8 bit processors, Rijndael algorithm can be programmed in assembly language with straightforward implementation for the *ShiftRows* and *AddRoundKeys* steps. The Rijndael algorithm can also be implemented very efficiently on processors 32 bit words with using four tables for the main rounds. As described in detail in the algorithm specification [6], each column in the output state can be computed using four exclusive OR instructions, a 32 bit word from the key schedule and four 32 bit words from the tables that are indexed using four bytes from the input state. Since the last round of Rijndael does not have *MixColumns* step a different set of tables would also have to be implemented.

Chapter 3: Requirements

3.1 Introduction

This Chapter identifies and discusses the key requirements needed for the assembly and high-level language software implementations of the AES (Rijndael) algorithm on the different platforms. The requirements were gathered from the detailed discussion of the functionality of the AES algorithm in the literature review (Chapter 2) and existing software implementations of the algorithm. The requirements are divided into two sets namely, *functional* and *non-functional* requirements.

3.2 Functional Requirements

The *functional* requirements describe the actual required functionality of the software implementations. Although very efficient source code of the Rijndael in various high-level programming languages, for example Dr.Gladman's C/C++ was available for use, it was felt that high-level language software implementation for this project needed to be written from scratch to gain better understanding of the algorithm. It was decided that though the already existing efficient high-level language implementations would provide better performance results than a simplified "clean" implementation, the results would be incomparable to those of the assembly language implementation.

The reasoning being that, the added complexity and functionality in the exiting high-level language implementations was beyond the scope of this project and given that performance of both high-level and assembly language software implementations needed to be compared, there had to be consistency in how the functionality of the Rijndael algorithm was implemented in both implemented software programs.

3.2.1 Software Requirements

The following functionality was identified as being essential for the software implementations of the AES algorithm.

- The software implementations of the Rijndael algorithm should treat the input block, output block and cipher key as arrays of 8 bit bytes.

- The software implementations of the Rijndael algorithm should take into consideration of the endianness of processor for which it is being designed. This is to ensure that right directions that the right directions of *shifts* and *rotates* operations are employed for the processor. *Endianness* refers to how 32 bit integers (words) are stored in memory of a computer. On big-endian processors like SPARC and Motorola 68000, the most significant byte is stored first, while on little-endian processors like Intelx86 and DEC VAX the least significant byte is stored first. Given the following hexadecimal 32 bit integer 0xA0B70708, in the memory of a big-endian processor it will be stored as A0 B7 07 08, while on a little-endian processor it will be stored a 08 07 b7 A0.

- The software implementations of the AES (Rijndael) algorithm should use the test vectors provided in the appendix of the FIPS standard [7], to verify that the functionality of the implementation performs as it should.

- The software implementations of the AES algorithm should take as input 128 bits block and encrypt or decrypt it using 128 bit, 192 bit or 256 input cipher key and output a 128 bit block.

- The software implementations of the Rijndael algorithm in both the high-level language and assembly language should be implemented in similar way so that their performance is comparable.

- The programming language used for the high-level language implementation of the AES algorithm should be portable over different platforms. This is because the high-level implementation developed for this project will have to compiled and run on various platforms as stated in the objectives (see section 1.3).

- The performance software implementations of the AES algorithm in both the high-level language and assembly language should be measurable either in terms of execution times or clock cycles.

3.3 Non-functional Requirements

The *non-functional* requirements describe the constraints placed on the development of the project. Firstly, given that 8 bit platforms were not readily available, an assembler/simulator would need to be used to assemble and run the assembly language implementation on a Windows PC.

The software development for this project will need to be done fairly quickly to allow time for the performance testing and write up of the dissertation. With the time constraint placed upon the development of the software, the components of the algorithm will need to be tested as immediately to eradicate the problem of debugging an already complex implementation. Also due to limited time for the development of the software, the high-level and assembly language of choice for the development of this project would need to be familiar or easy to learn and implement.

The hardware resources that were available for the development and testing of this project are the university computers provided by BUCS and a personal computer.

3.4 Requirement Specification

As stated in the Introduction (Chapter 1), the overall aim of this dissertation is to measure the performance of software implementations of the AES (Rijndael) algorithm on the different platforms. Two Rijndael algorithm software implementations would have to be developed, one in a preferred assembly language and another in a high-level programming language. The assembly language implementation would need to run and assemble on a chosen microprocessor simulator. The choice of simulator/emulator depended mainly on availability and the ease with which the Rijndael algorithm would be implemented in its assembly language given the time constraints on the project. The Antonakos' 68000 microprocessor simulator with ASM68K assembler and EMU68K emulator was the choice for this project. Although 68000microprocessor is 16 bit platform, it is still representative for the microprocessors used in various devices today like smart cards (debit/credit) and automatic bank tellers that the 8 bit platform was meant to represent.

The high-level language software implementation of the AES algorithm will also developed to run on the 32 bit, 64 bit and multi-processor platforms which are representative for personal computers and high-end workstations (servers). C++ was the programming language of choice for the high-level language mainly because of its portability over different platforms. The reasoning behind the choice of the 68000 assembly language and C++ for the software implementations for this project is discussed in detail in the next section.

Chapter 4: Design

4.1 Overview

A high level overview of the solution for this project is presented in section. It serves to document the approach and the key choices for made whilst producing the software implementations of the Rijndael algorithm. This includes the justification for the programming languages chosen for the implementations of the AES algorithm, followed by an outline of the overall architecture of the project and a detailed look at the different methods for implementing the key components of the algorithm along with high-level design of the AES implementations.

4.2 Choice of Programming languages

As stated earlier in the requirements specification (section 3.4), standard C++ and the 68000 assembly language were the choices for the high-level and assembly language implementations respectively for this project. These programming languages were preferred mainly due to benefits they offered to software implementation of the AES algorithm and other general reasons both outlined below:

- The programming languages of choice for this project needed to be familiar or easy to learn, as the time constraint on the software development phase of the project needed to be taken into consideration. With only a few months to write two programs in two different languages, there was no time or benefit of learning an unfamiliar or complicated language.

- Compiled high-level languages like C++ are generally better adapted to optimizing performance than interpreted language like Java. A program translated by a complier is often much faster than an interpreter executing the same program. Therefore a software implementation of Rijndael algorithm in standard C++ would produce better performance results than say a Java implementation.

- Given that, there already existed C++ implementation of the Rijndael algorithm this reduced the time needed for the development of the project.

- Internally the AES algorithm operates on a two dimensional array of bytes called the state that contains 4 rows and 4 columns. Each column or of the state can be thought of as an array of four bytes thus a 32 bit word. The 68000 microprocessor even though is a 16 bit processor, has address and data registers that are 32 bit wide and therefore can perform a wide variety of operations on 32 bit data words. This would increase the efficiency of the software implementation of the Rijndael algorithm in the 68000 assembly language.

- The free availability and ease use of the two IBM-PC executable programs, ASM68K (a 68000 assembler) and EMU68K (a 68000 emulator) made the 68000 assembly language the straightforward choice for this project. Finally also due to the availability of Microsoft Visual C++ software on the BUCS computer and the author's personal computer, coupled with the above mentioned benefits, C++ became the obvious choice for high-level language implementation for this project.

4.3 Overall Architecture

Rijndael algorithm consists of four byte oriented transformations (as shown in figure 4.1) which can be thought of a functions in a high-level programming language. In an object oriented programming language like C++ which is used for this project, the encryption and decryption modules can be thought of as member functions of the main class. Modularising the algorithm in such a way makes the maintenance and understanding of the software implementations easier.

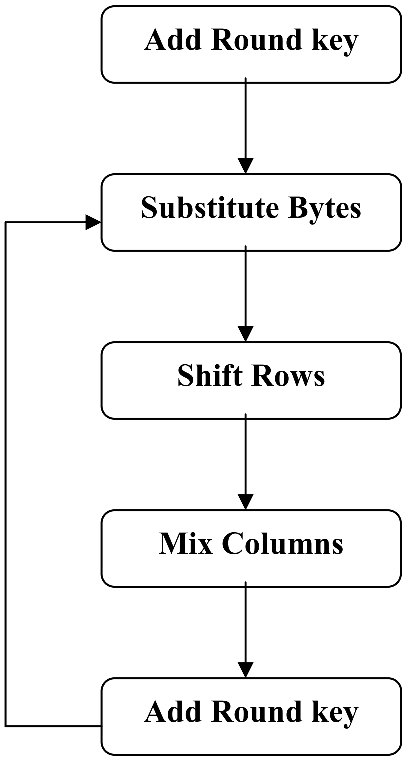


Figure 4.1: The Rijndael algorithm components

The AES (Rijndael) algorithm is a key symmetric algorithm, where both the encryption and decryption processes use the key. In the AES (Rijndael) encryption process the input plaintext is broken into 16 byte (128 bits) blocks. Each 128 bits block is then combined with the encryption key using the algorithm performs various byte transformations resulting in the production of an encrypted block. The decryption process takes the encrypted block plus the same encryption key using the algorithm performs inverse byte transformations resulting in a plaintext block.

4.3.1 Key Scheduling

The software implementations of the AES (Rijndael) algorithm would have to support three different input key sizes used by algorithm. The different key sizes that may be used by the algorithm are:

- 16 byte -128 bits key
- 24 byte -192 bits key
- 32 byte -256 bits key

In order to prepare for the round transformations described below, a key expansion operation called the key schedule (see section 2.4) is executed. This operation uses original input key to create several round keys. These round keys including the original input key would be used in each round of the decryption or encryption process. Figure 4.2 illustrates the key schedule and round key selection for the Rijndael algorithm, where w_i represents the individual 32 bit word keys which make up the round key represented by $k\left[i\right]$. In the case where the original input key is 128 bits (16 bytes), the round key $k\left[0\right]$ would be filled by the input key and the next round key $k\left[1\right]$ would be generated from the last 32 bit word key w_3 of the input key.

w₀	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}

round key k[0]	***round key k[1]***	***round key k[2]***	***round key k[3]***

Figure 4.2: The key Schedule and round key selection for AES algorithm

This process of expanding the input key will continue until enough 32 bit word keys are created to cover the number of rounds plus the initial *AddRoundKeys* step. The total numbers of 32 bit word keys need for the variations of the AES algorithm are shown in the table below:

Table 4.1: Total number of key words

Key bits	Number of words
128	44
192	52
256	60

There are two kinds of implementation for key scheduling- key unrolling and key on-the-fly. For the key unrolling implementation all the needed round keys are generated and stored in memory at the decryption or encryption process, while for the key on-the-fly implementation the keys are created just before they are used. The on-the-fly key schedule requires less memory which could be advantage on Smart card implementations of the Rijndael algorithm. For this project the key unrolling scheduling was used, the reasoning being that it offered better performance and with the encryption and decryption process using the same key, it was best to have one function for the key scheduling.

4.3.2 Input and Output Considerations

The input and output of AES (Rijndael) algorithm each consist of sequences of 128 bits (binary digits 0 and 1). This sequence is referred to as a block with length of 128 bits. The bits within the block are numbered starting at zero and ending at one less than the block length. The number i associated with a bit within the sequence will therefore be in the range $0 \le i \le 128$. The basic unit of processing in the Rijndael algorithm is a byte, a sequence of eight bits treated as a single entity (see section 2.2.1). For the software implementations of the algorithm in this project the bit sequences are mapped onto arrays of 8-bit bytes with consecutive bytes formed from consecutive 8-bit sub-sequences.

Internally the Rijndael algorithm operates on a two dimensional array of bytes called the *state* as described in section 2.2.1. The state for the AES consists of 4 rows and 4 columns because the block length is fixed to 128 bits. In this state array, denoted by the symbol s, each individual byte has two indexes: its row number r, in the range $0 \le r \le 4$, and its column number c, in the range $0 \le c \le 4$. This allows individual bytes in state to be referred to either as $s_{r,c}$ or $s[r,c]$.

At the start of the encryption or decryption process the bytes of the input block are copied to the state array as illustrated by Fig.4.3 below. The encryption or decryption operations are then applied to the bytes of the state, after which the output decrypted or encrypted block is produced.

$$\begin{array}{c}\textit{input block}\\ \boxed{\begin{array}{cccc} in_0 & in_4 & in_8 & in_{12} \\ in_1 & in_5 & in_9 & in_{13} \\ in_2 & in_6 & in_{10} & in_{14} \\ in_3 & in_7 & in_{11} & in_{15} \end{array}}\end{array} \rightarrow \begin{array}{c}\textit{state array}\\ \boxed{\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{array}}\end{array} \rightarrow \begin{array}{c}\textit{output block}\\ \boxed{\begin{array}{cccc} out_0 & out_4 & out_8 & out_{12} \\ out_1 & out_5 & out_9 & out_{13} \\ out_2 & out_6 & out_{10} & out_{14} \\ out_3 & out_7 & out_{11} & out_{15} \end{array}}\end{array}$$

Figure 4.3: Input, Output and State array

Hence as described in the FIPS standard [7], at the start of the encryption or decryption the input block is copied to the state array according to the following scheme:

$$s[r,c] = in[r+4c] \quad \text{for } 0 \le r \le 4 \text{ and } 0 \le c \le 4,$$

and the state array is copied to the output block using the following scheme:

$$out[r+4c] = s[r,c] \quad \text{for } 0 \le r \le 4 \text{ and } 0 \le c \le 4$$

For this project the software implementations of the Rijndael algorithm interpreted the state array as one-dimensional array of four 32 bit words. The four bytes in the in each the columns of the state array shown in Fig: 4.3 forming a 32 bit word.

The number of rounds performed during the encryption or decryption process of the AES algorithm, are dependant on the input key size (see section 2.2.2). The table 4.2 below shows the number of rounds performed by the software implementations developed for this project.

Table 4.2: Number of AES rounds

Key size in bits	**Block size in bits**	**Number of rounds**
128	128	10
192	128	12
256	128	14

The round function for both the encryption and decryption AES algorithm have four different byte-oriented transformations as described in literature review (section 2.2). These byte operations transform the state array by substituting the bytes within state array with bytes in a table, shifting the rows of the state array by different offsets, mixing the bytes within each column of the state and adding a round key to the state.

4.3.3 Encryption Round Transformations

In the encryption process each of the rounds of the Rijndael algorithm (with exception of the final round) executes the following transformations in the sequence that they are shown below:

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKeys

The final round of the does not have the *MixColumns* transformation, therefore executes the transformations in the following sequence.

- SubBytes
- ShiftRows
- AddRoundKeys

SubBytes

In this transformation, each byte of the state is replaced by the content of the encryption substitution table shown below (Table 4.3) at the position defined by the byte that is going to be substituted. For example if a state byte, $s_{0,0} = \{8d\}$ is the byte being substituted, then in the Substitution table go to 8 the $'x'$ axis (vertical axis) and d in the $'y'$ axis (horizontal axis) to get the substituted valve of state byte, $s'_{0,0} = \{5d\}$.

Table 4.3: The Encryption Substitution Table (in hexadecimal)

[xy]	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	C3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	B5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

The encryption Substitution table which is referred to the S-box in the FIPS [7], is the final result of two stage transformation described in section 2.2.3. In the software implementations of the AES algorithm the S-box could seen as a 256 byte array and the *SubBytes* transformation would be performed by table lookup.

ShiftRows

The second transformation of Rijndael algorithm cyclically shifts the rows of the state except the row numbered 0 (see section 2.2.4). For the software implementation of AES algorithm the *ShiftRows* transformation is equivalent to rotating left the bytes in the rows numbered 1, 2 and 3 of the state once, twice and three times respectively. This has the effect of moving bytes to lower positions in the row, while the lowest bytes wraps around into the top of the row.

MixColumns

The *MixColumns* transformation operates the state column by column, treating each column as a four degree polynomial as described in section 2.2.5. This can be written as the matrix multiplication shown below:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c \le 4$$

After this multiplication, the bytes in the in the above column are replaced as follows:

$$\begin{aligned} s'_{0,c} &= \{02\} \bullet s_{0,c} \oplus \{03\} \bullet s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= \{02\} \bullet s_{1,c} \oplus \{03\} \bullet s_{2,c} \oplus s_{3,c} \oplus s_{0,c} \\ s'_{2,c} &= \{02\} \bullet s_{2,c} \oplus \{03\} \bullet s_{3,c} \oplus s_{1,c} \oplus s_{0,c} \\ s'_{3,c} &= \{02\} \bullet s_{3,c} \oplus \{03\} \bullet s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \end{aligned}$$

Where $\bullet$ and $\oplus$ represent finite field multiplication and addition (see section 2.1.4) respectively. But since $\{03\} \bullet s_{0,c} = \{02\} \bullet s_{0,c} \oplus s_{0,c}$, the bytes in the column above can also be written as below, where $v = s_{3,c} \oplus s_{2,c} \oplus s_{1,c} \oplus s_{0,c}$.

$$\begin{aligned} s'_{0,c} &= v \oplus s_{0,c} \{02\} \bullet (s_{1,c} \oplus s_{0,c}) \\ s'_{1,c} &= v \oplus s_{1,c} \{02\} \bullet (s_{2,c} \oplus s_{1,c}) \\ s'_{2,c} &= v \oplus s_{2,c} \{02\} \bullet (s_{3,c} \oplus s_{2,c}) \\ s'_{3,c} &= v \oplus s_{3,c} \{02\} \bullet (s_{0,c} \oplus s_{3,c}) \end{aligned}$$

For the software implementation of *MixColumns* transformation the above formulation is quite efficient on 8 bit processor [6], given that the finite field multiplication by the element $\{02\}$ is shift followed by an exclusive OR operation.

AddRoundKeys

In the last transformation *AddRoundKeys*, a round key is added to the state by a simple bitwise exclusive OR operation as illustrated Fig.4.4 below:

$$\boxed{\begin{matrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{matrix}} \oplus \boxed{\begin{matrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{matrix}} = \boxed{\begin{matrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{matrix}}$$

Figure 4.4: The state array exclusive OR-ed with a round key

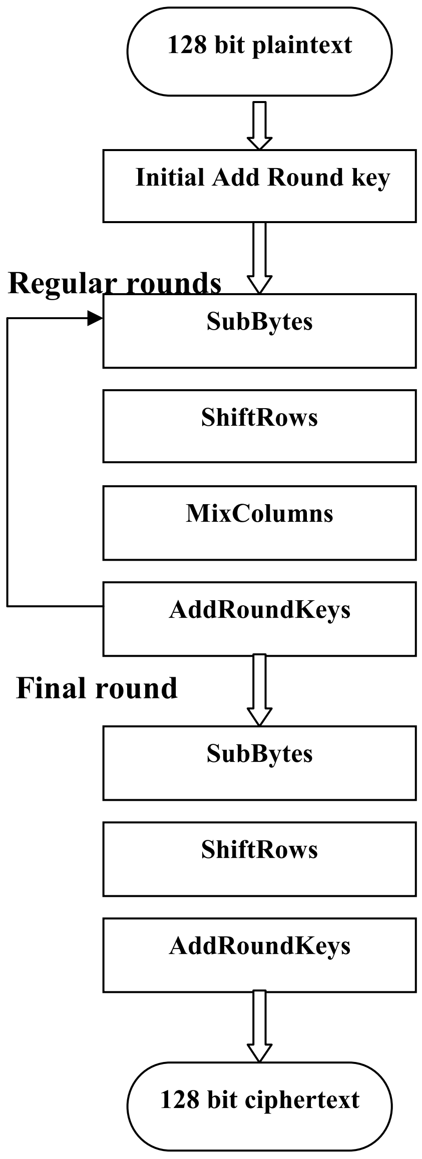


Figure 4.5: AES Encryption flowchart

4.3.4 Decryption Round Transformations

In the decryption process each of the rounds of the Rijndael algorithm (with exception of the final round) executes the following transformations in the sequence that they are shown below:

- Inverse SubBytes
- Inverse ShiftRows
- Inverse MixColumns
- AddRoundKeys

The final round of the does not have the *inverse MixColumns* transformation, therefore executes the transformations in the following sequence.

- Inverse SubBytes
- Inverse ShiftRows
- AddRoundKeys

Inverse SubBytes

The *inverse SubBytes* transformation operates on the state in exactly the same way that the *SubBytes* transformation does (see section 4.3.3) using the decryption substitution table below:

Table 4.4: The Decryption Substitution Table (in hexadecimal)

[xy]	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Inverse ShiftRows

The *inverse ShiftRows* transformation of Rijndael algorithm cyclically shifts the rows of the state except the row numbered 0 as shown Fig: 4.6. For the software implementation of AES algorithm the *inverse ShiftRows* transformation is equivalent to rotating right the bytes in the rows numbered 1, 2 and 3 of the state once, twice and three times respectively. This has the effect of moving bytes to higher positions in the row, while the highest bytes wraps around into the bottom of the row.

$$\boxed{\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{array}} \rightarrow \boxed{\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,3} & s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,1} & s_{3,2} & s_{3,3} & s_{3,0} \end{array}}$$

Figure 4.6: Inverse ShiftRows transformation

Inverse MixColumns

The *inverse MixColumns* transformation operates the state column by column, treating each column as a four degree polynomial as described in section 2.3. This can be written as the matrix multiplication shown below:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c \le 4$$

After this multiplication, the bytes in the in the above column are replaced as follows:

$$\begin{aligned} s'_{0,c} &= \{0e\} \bullet s_{0,c} \oplus \{0b\} \bullet s_{1,c} \oplus \{0d\} \bullet s_{2,c} \oplus \{09\} \bullet s_{3,c} \\ s'_{1,c} &= \{0e\} \bullet s_{1,c} \oplus \{0b\} \bullet s_{2,c} \oplus \{0d\} \bullet s_{3,c} \oplus \{09\} \bullet s_{0,c} \\ s'_{2,c} &= \{0e\} \bullet s_{2,c} \oplus \{0b\} \bullet s_{3,c} \oplus \{0d\} \bullet s_{0,c} \oplus \{09\} \bullet s_{1,c} \\ s'_{3,c} &= \{0e\} \bullet s_{3,c} \oplus \{0b\} \bullet s_{0,c} \oplus \{0d\} \bullet s_{1,c} \oplus \{09\} \bullet s_{2,c} \end{aligned}$$

Where $\bullet$ and $\oplus$ represent finite field multiplication and addition (see section 2.1.4) respectively. As described in detail in the Rijndael algorithm specification [6], the bytes in the column above can also be written as below, where $v = \{09\} \bullet \left(s_{3,c} \oplus s_{2,c} \oplus s_{1,c} \oplus s_{0,c}\right)$

$$\begin{aligned} s'_{0,c} &= \left(\{04\} \bullet \left(s_{2,c} \oplus s_{0,c}\right)\right) \oplus \left[v \oplus s_{0,c} \oplus \{02\} \bullet \left(s_{1,c} \bullet s_{0,c}\right)\right] \\ s'_{1,c} &= \left(\{04\} \bullet \left(s_{3,c} \oplus s_{1,c}\right)\right) \oplus \left[v \oplus s_{1,c} \oplus \{02\} \bullet \left(s_{2,c} \bullet s_{1,c}\right)\right] \\ s'_{2,c} &= \left(\{04\} \bullet \left(s_{2,c} \oplus s_{0,c}\right)\right) \oplus \left[v \oplus s_{2,c} \oplus \{02\} \bullet \left(s_{3,c} \bullet s_{2,c}\right)\right] \\ s'_{3,c} &= \left(\{04\} \bullet \left(s_{3,c} \oplus s_{1,c}\right)\right) \oplus \left[v \oplus s_{3,c} \oplus \{02\} \bullet \left(s_{0,c} \bullet s_{3,c}\right)\right] \end{aligned}$$

As for the software implementation of *MixColumns* transformation, the above *inverse MixColumns* formulation can implemented using the finite field multiplication by the element $\{02\}$, which is shift followed by an exclusive OR operation

AddRoundKeys

The *AddRoundKeys* transformation of the decryption process of the AES algorithm is exactly the same operation as the one of encryption process illustrated in Fig.4.4 above. The only difference being that the round keys from the key scheduling are used in reverse order (see section 2.3).

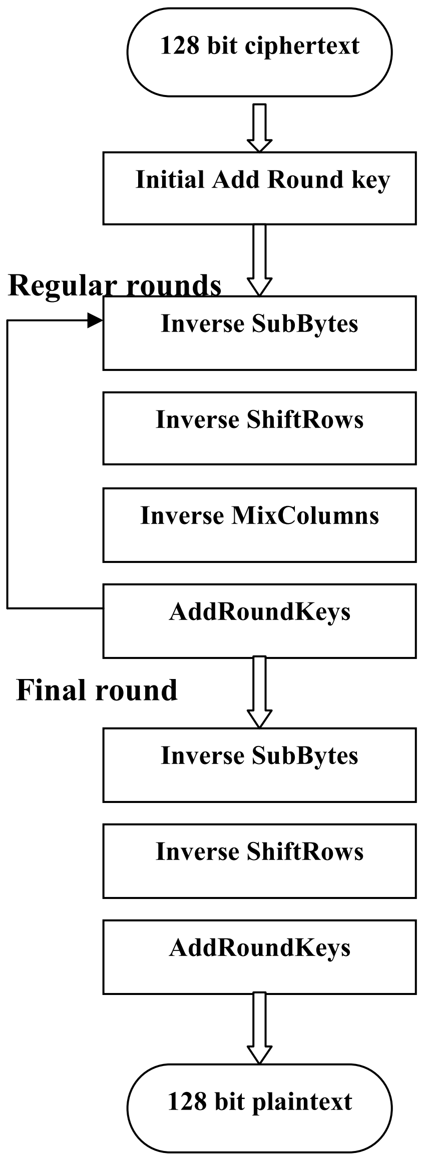


Figure 4.7: AES Decryption flowchart

4.4 Summary

This chapter presented a high level overview of the Rijndael algorithm implementations for this project. Decisions about which programming languages should be used for the implementations as well as the design and methods of implementing the different functions that make up the algorithm have been discussed. The encryption and decryption processes discussed in this chapter should help the reader follow and understand the details of the AES (Rijndael) algorithm implementations for this project, which are discussed in the next chapter.

Chapter 5: Implementation

5.1 Introduction

This section provides the details of the software implementations of the Rijndael algorithm that were developed for this dissertation. Throughout this chapter we will also apply the design solutions to the key functions of the algorithm that have been specified in the previous chapter.

5.2 Platforms

For this project, standard C++ and 68000 assembly languages were the languages of choice for the implementation of the AES (Rijndael) algorithm (see section 4.2). The C++ implementation was developed in Microsoft Visual C++ (version 6) software, which provides an excellent workspace feature for the efficient management of the several **.cpp** files need for algorithm implementation. The other advantage of using Microsoft's Visual C++ software for this project is its Integrated Development Environment (IDE) compiling option that helpful when debugging a program.

The 68000 assembly programs were written in a text editor, in this case Microsoft's Windows notepad and saved as **.asm** files that were later assembled by the 68000 assembler program ASM68K. The ASM68K assembler program takes a 68000 assembly source file with extension **.asm** and determines the machine language for each of source statements. The EMU68K emulator program which simulates the operations of the 68000 microprocessor takes the object file with extension **.hex** produced by the assembler and runs it.

5.3 Software Developed

The Software implementations for this project were developed in separate files for decryption and encryption processes of the Rijndael algorithm. The files below outline the standalone functionality provide by the C++ source code (see Appendix A) developed for this project in Microsoft Visual C++.

- `aes.h`

  This is the main C++ header file that contains the main class interface `Rijndael_class`, for the encryption and decryption implementations. This file also contains the global variables, constants and the pre-processor directives.

- `aes_encrypt.cpp`

  This is the source code for the C++ Rijndael algorithm encryption implementation.

- `aes_decrypt.cpp`

  This is the source code for the C++ Rijndael algorithm decryption implementation.

- `aes_main.cpp`

  This is the `main()` C++ program where the execution of the C++ implementation begins.

- `aes_key.cpp`

  This is the C++ source code for the key scheduling for both the encryption and decryption implementations.

For the 68000 assembly language implementation of AES algorithm, it was decided that separate assembly source code (shown in Appendix B) would be written for each of the key variations of the algorithm. By specifying each of these files outlined below as separate entities we gained the advantages of being able to copy and reuse main aspects of them, as well as easy readability and maintenance of assembly code.

- `aes128.asm` – This is the 68000 assembly language encryption implementation of the Rijndael algorithm for the128 bit input key and 128 bit block.

- `aes192.asm` – This is the 68000 assembly language encryption implementation of the Rijndael algorithm for the192 bit input key and 128 bit block.

- `aes256.asm` – This is the 68000 assembly language encryption implementation of the Rijndael algorithm for the 256 bit input key and 128 bit block.

- `Inv128.asm` – This is the 68000 assembly language decryption implementation of the Rijndael algorithm for the128 bit input key and 128 bit block.

- `Inv128.asm` – This is the 68000 assembly language decryption implementation of the Rijndael algorithm for the128 bit input key and 128 bit block.

- `Inv128.asm` – This is the 68000 assembly language decryption implementation of the Rijndael algorithm for the128 bit input key and 128 bit block.

Unlike the 68000 assembly implementation of the AES which was entirely written by the author, the C++ implementation re-used for some of its functionality, source code from exiting implementations in the public domain. This code was retested and adapted to meet the requirements of this project.

The C++ implementation was developed in such way that it could function correctly on any processor regardless of its endianness as required by the requirements specification (see section 3) for this project. This was achieved by converting the input block, which in this case is a 32 digit hexadecimal number into the 16 bytes with the bits ordered correctly before storing them in the state array.

5.4 Overall Architecture

Here we will provide a more detailed description of how the AES (Rijndael) algorithm was implemented for this project. The details of how round transformations of the encryption and decryption processes and the key scheduling of the Rijndael algorithm (see section 4.3) were implemented in 68000 assembly language and C++ are described with the help of pseudo code.

In the pseudo code in this section the following symbols will be used:

& - bitwise logical AND operation between two operands

$ - stands for a hexadecimal value in 68000 assembly language

0x – stands for a hexadecimal value in C++

^ - bitwise exclusive OR operation between two operands

| - bitwise logical OR operation between two operands

<< - logical shift left operation

>> - logical shift right operation

Nk - number of 32 bit words comprising the input key. For this project ***Nk*** = 4, 6 or 8.

Nb - number of 32 bit words comprising the input block. For this project ***Nb*** = 4.

Nr - number of encryption or decryption rounds. For this project ***Nr*** = 10, 12 or 14.

5.4.1 The State

The Rijndael state as described in section 4.3.2 is a two dimensional array of bytes, made up of four 32 bit words. For the C++ implementation of the AES algorithm the state was treated a 4 × 4 two-dimensional array `unsigned char state[4][4]`. In C++ each `unsigned char` element occupies a byte therefore the state two-dimensional array provided 16 bytes (128 bits) as needed input block.

In the 68000 assembly language implementation, the Rijndael state was stored in memory as four consecutive 32 bit words using the assembly language instruction below:

```
BLOCK DC.L $3243f6a8, $885a308d, $313198a2, $e0370734
```

Where `BLOCK` is one-dimensional array with four long words each representing a 32 bit word or column of the state.

5.4.2 Key Expansion

For this project the input key expansion of the Rijndael algorithm was implemented using key unrolling scheduling (see section 4.3.1). This means that the input key was expanded to cover the number of required 32 bit words for the round transformations and stored in memory. In the case of C++ implementation expanded key is stored in a three dimensional array `unsigned char key_buffer[15][4][4]`, where a total of fourteen 16 byte keys can be stored. In each of the 68000 assembly implementations the expanded key is stored in memory using the assembly instruction as shown below:

```
KEY_BUF    DS.L    44
```

Where the assembler directive `DS.L` (Define Storage) creates storage for 44 long words (32 bit words) needed for the 128 bits input key AES implementation.

The following pseudo code describes how the key scheduling was implemented for this project:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1), Nk)
begin
     word temp

     i = 0

      while (I < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2],
                key[4*i+3])

        i = i+1
      end while

     i = Nk

     while (I < Nb * (Nr+1))
       temp = w[i-1]
       if (i mod Nk = 0)
           temp = subWord(RotWord(temp))^ Rcon[i/Nk]
       else if (Nk > 6 and i mod Nk = 4)
           temp = subWord(temp)
       end if
       w[i] = w[i-Nk]^ temp
       i = i + 1
     end while
```

```
end
```

subWord() is a function that a four byte input word and applies the *SubBytes* transformation (see section 4.3.3) on each of the bytes of the four bytes. **RotWord()** is a four byte input word and performs a cyclic left shift. **Rcon[]** is an array which contains the round-dependant constants (see section 2.4). **key[]** is the array containing the input key bytes and **w[]** is the array where the 32 bit word keys are stored.

5.4.3 Cipher Transformations

At the start of the encryption process as described in section 4.3.3, the input block (plaintext) is copied into the state as described in section 5.4.1. After the initial round key addition, the state is transformed by the round function 10, 12 or 14 times depending on the size of the input key. The final round differs from the regular rounds as demonstrated in Fig.4.5, since it does not include the *MixColumns* transformation. The final state is then copied to the output as ciphertext as demonstrated by Fig.4.5.

The pseudo code below describes how the encryption process was implemented in the Rijndael algorithm software developed for this project (see Appendix for source code written).

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte state[4,Nb]

   state = in

   AddRoundKeys(state, w[0, Nb-1])

for round = 1 step 1 to Nr-1

   SubBytes(state)
   ShiftRows(state)
   MixColumns(state)
   AddRoundKeys(state, w[round*Nb, (round+1)*Nb-1)
end for

  SubBytes(state)
  ShiftRows(state)
  AddRoundKeys(state, w[Nr*Nb, (Nr+1)*Nb-1)

  Out = state
end
```

The array **w[]** contains the key schedule described in section 5.4.2. The individual transformations (see section 4.3.3) used in the Rijndael encryption algorithm implementations developed for this dissertation are described next.

SubBytes()

The *SubBytes* transformation as described in section 4.3.3 was implemented using the substitution table. In both the 68000 assembly language and C++ Rijndael algorithm implementations developed for this dissertation, the encryption substitution table (S-box) was stored in 256 byte array. The lookup table implementation of the *SubBytes* transformation offers speed gains, since retrieving a value from memory is often faster than undergoing an expensive computation. The following pseudo code describes how this transformation was implemented:

```
SubBytes(byte state[4,Nb], Nb)
begin
  for r = 0 step 1 to 3
    for c = 0  step 1 to Nb-1
       state[r,c] = S-box[state[r,c]]
    end for
  end for
end
```

ShiftRows()

The *ShiftRows* transformation (see section 4.3.3) cyclically left shifts each of the last three rows of state 1, 2 and 3 time respectively. The pseudo code below describes how this transformation was implemented for the software developed for this project.

```
ShiftRows(byte state[4,Nb], Nb)
begin
   byte t[Nb]
   for r = 1 step 1 to 3
     for c = 0 step 1 to Nb - 1
       t[c] = state[r, (c + h(r,Nb)) mod Nb]
     end for
     for c = 0 step 1 to Nb – 1
        state[r,c] = t[c]
     end for
   end for
end
```

MixColumns()

For this project the *MixColumns* transformation formulation as described in section 4.3.3 was implemented in C++ using this further optimized code sequence that uses the temporary variables `t, u and v`. The *MixColumns* transformation operates on each column (32 bit word) of the state as follow:

$$
\begin{aligned}
&t = s_{3,c} \wedge s_{2,c} \\
&u = s_{1,c} \wedge s_{0,c} \\
&v = t \wedge u \\
&s'_{0,c} = s_{0,c} \wedge v \wedge FFmul(u) \\
&s'_{1,c} = s_{1,c} \wedge v \wedge FFmul(0\text{x}02, s_{2,c} \wedge s_{1,c}) \\
&s'_{2,c} = s_{2,c} \wedge v \wedge FFmul(0\text{x}02, t) \\
&s'_{3,c} = s_{3,c} \wedge v \wedge FFmul(0\text{x}02, s_{0,c} \wedge s_{3,c})
\end{aligned}
$$

`FFmul()` is a function that performs finite field multiplication, which in the case of the C++ implementation for this project was achieved by repeated shifts (see section 2.1.4) and it was specified as an inline function to make the finite field multiplication more efficient.

An inline function is small function in C++ whose code is expanded in line rather than called. When a function is called, a series of instructions must be executed, both to set up the function call and to return from the function. In some cases, many CPU cycles are used to perform these actions. However, when a function is expanded in line, no such overhead exists, and the overall performance of the program increases.

The 68000 assembly language implementation of the *MixColumns* transformation instead of using the above code sequence where operations are performed on every byte of each column of the state, the code sequence was rewritten to operate on each column as whole (32 bit word). Each column `w` (32 bit word) of the state is operated on by the following code sequence to achieve the *MixColumns* transformation.

```
W'= rot1(w) ^ rot2(w) ^ rot3(w)^ FFmulX(w ^ rot3(w))
```

`rot1(w)`- is equivalent to rotate right 8 bits the word `w` in 68000 assembly language.

`rot2(w)`- is equivalent to rotate right 16 bits the word `w` in 68000 assembly language.

`rot3(w)`- is equivalent to rotate left 8 bits the word `w` in 68000 assembly language.

`FFmulX(w)`- performs a finite field multiplication on each of the four bytes in the word `w` with hexadecimal value 0x02 or $\{02\}$. The pseudo code below describes how this finite field multiplication was implemented for this project.

```
Word FFmulX(const word w)
begin
   word t = W & 0x80808080
   return ((w ^ t)<<1) ^ ((t>>3)|(t>>4)|(t>>6)|(t>>7)
end
```

This *MixColumns* transformation developed for 68000 assembly language implementation offers performance benefits on processors that have operations that can cyclically rotate bytes within a 32 bit word like the 68000 microprocessor.

`AddRoundKeys()`

This transformation is a simply a bitwise exclusive OR operation between the state and a round key (see section 4.3.3). For the 68000 assembly implementation the *AddRoundKeys* transformation was achieved by exclusive OR-ing the 32 bit words of the state with 32 bit words of the key round. The pseudo code below describes how this transformation was implemented in C++.

```
AddRoundKeys(byte state[4, Nb], byte key[4, Nb],Nb)
begin
 for r = 0 step 1 to Nb - 1
   for c = 0 step 1 to 3
        state[r,c] = state[r,c] ^ key[r,c]
   end for
 end for
end
```

5.4.4 Inverse Cipher Transformations

At the beginning of the decryption process as described in section 4.3.4, the input block (ciphertext) is copied into the state as described in the encryption process above. After the initial round key addition, the state is transformed by the round function 10, 12 or 14 times depending on the size of the input key. The final round differs from the regular rounds as demonstrated in Fig.4.7, since it does not include the *inverse MixColumns* transformation. The final state is then copied to the output as plaintext as demonstrated by Fig.4.7.

The pseudo code below describes how the decryption process was implemented in the AES software developed for this project (see Appendix for source code written).

```
InvCipher(byte    in[4*Nb],    byte    out[4*Nb],    word
w[Nb*(Nr+1)])

begin
   byte state[4,Nb]

   state = in

   AddRoundKeys(state, w[Nr*Nb, (Nr+1)*Nb-1)

for round = Nr-1 step -1 to 1

   Inv_ShiftRows(state)
   Inv_SubBytes(state)
   AddRoundKeys(state, w[round*Nb, (round+1)*Nb-1)
   Inv_MixColumns(state)
end for

  Inv_ShiftRows(state)
  Inv_SubBytes(state)
  AddRoundKeys(state, w[0, Nb-1])

  Out = state
end
```

The array **`w[]`** contains the key schedule described in section 5.4.2 since the same key is used for both the cipher and inverse cipher but in reverse order. The individual transformations (see section 4.3.4) used in the Rijndael decryption algorithm implementations developed for this dissertation are described next.

`Inv_ShiftRows()`

The inverse *ShiftRows* transformation (see section 4.3.4) which cyclically right shifts each of the last three rows of state 1, 2 and 3 time respectively is the first round transformation in the decryption process. For Rijndael algorithm implementations developed for this project, the pseudo code below describes how this transformation was implemented.

```
Inv_ShiftRows(byte state[4,Nb], Nb)
begin
   byte t[Nb]
   for r = 1 step 1 to 3
     for c = 0 step 1 to Nb - 1
       t[(c + h(r,Nb)) mod Nb] = state[r,c]
     end for
     for c = 0 step 1 to Nb - 1
         state[r,c] = t[c]
     end for
   end for
end
```

`Inv_SubBytes()`

The inverse *SubBytes* transformation as described in section 4.3.4 was implemented using the substitution table. In both the 68000 assembly language and C++ Rijndael algorithm implementations developed for this project, the decryption substitution table (Inv_S-box) is stored in 256 byte array. The table lookup implementation that was used for the inverse *SubBytes* transformation is similar to the one used for the *SubBytes* transformation (see section 5.4.3). The pseudo code below describes how this transformation was implemented for this project.

```
Inv_SubBytes(byte state[4,Nb], Nb)
begin
  for r = 0 step 1 to 3
    for c = 0  step 1 to Nb-1
       state[r,c] = Inv_S-box[state[r,c]]
    end for
  end for
end
```

AddRoundKeys()

The *AddRoundKeys* transformation is its own inverse, therefore the implementation is exactly the same as that described in section 5.4.3.

Inv_MixColumns()

The inverse *MixColumns* transformation formulation as described in section 4.3.4 was implemented in C++ using this further optimized code sequence that uses temporary variables `t, u, w and v`. The inverse *MixColumns* transformation operates on each column (32 bit word) of the state as follow:

$$
\begin{aligned}
&t = s_{3,c} \wedge s_{2,c} \\
&u = s_{1,c} \wedge s_{0,c} \\
&v = t \wedge u \\
&v = v \wedge FFmul(0x08, v) \\
&w = v \wedge FFmul\left(0x04, s_{2,c} \wedge s_{0,c}\right) \\
&v = v \wedge FFmul\left(0x04, s_{3,c} \wedge s_{1,c}\right) \\
&s'_{0,c} = s_{0,c} \wedge v \wedge FFmul\left(0x02, u\right) \\
&s'_{1,c} = s_{1,c} \wedge v \wedge FFmul\left(0x02, s_{2,c} \wedge s_{1,c}\right) \\
&s'_{2,c} = s_{2,c} \wedge v \wedge FFmul\left(0x02, t\right) \\
&s'_{3,c} = s_{3,c} \wedge v \wedge FFmul\left(0x02, s_{0,c} \wedge s_{3,c}\right)
\end{aligned}
$$

FFmul() is a function that performs finite field multiplication, which in the case of the C++ implementation for this project was achieved by repeated shifts (see section 2.1.4) and it was specified as an inline function to make the finite field multiplication more efficient.

The 68000 assembly language implementation of the inverse *MixColumns* transformation instead of using the above code sequence where operations are performed on every byte of each column of the state, the code sequence was rewritten to operate on each column as whole (32 bit word). Each column w (32 bit word) of the state is operated on by the following code sequence to achieve the inverse *MixColumns* transformation.

```
                    'w = FFmulX8(w) ^ w

W'= FFmulX(w) ^ FFmulx4(w) ^ FFmulx8(w) ^ rot3(FFmulX(w))
    ^ rot2(FFmulx4(w)) ^ rot3('w)^ rot2('w) ^ rot1('w)
```

rot1(w)- is equivalent to rotate right 8 bits the word w in 68000 assembly language.

rot2(w)- is equivalent to rotate right 16 bits the word w in 68000 assembly language.

rot3(w)- is equivalent to rotate left 8 bits the word w in 68000 assembly language.

FFmulX(w)- performs a finite field multiplication on each of the four bytes in the word w with the hexadecimal value 0x02 or $\{02\}$. The pseudo code below describes how this finite field multiplication was implemented for this project.

```
Word FFmulX(const word w)
begin
   word t = W & 0x80808080
   return ((w ^ t)<<1) ^ ((t>>3)|(t>>4)|(t>>6)|(t>>7)
end
```

FFmulX4(w)- performs a finite field multiplication on each of the four bytes in the word w with the hexadecimal value 0x04 o$\{04\}$. This operation was implemented by performing **FFmulX(w)** twice on each word w (column) of the state.

FFmulX8(w)- performs a finite field multiplication on each of the four bytes in the word w with the hexadecimal value 0x08 o$\{08\}$. This operation was implemented by performing **FFmulX(w)** three times on each word w (column) of the state.

The inverse *MixColumns* transformations developed for 68000 assembly language and C++ implementations as shown above are more complicated than the *MixColumns* transformations developed for the encryption process. This definitely impacts negatively on the performance of the decryption process.

5.5 Summary

This section has presented a detailed description of how the Rijndael algorithm was implemented in 68000 assembly language and C++ for this project. Also discussed are how the various requirements that were specified in chapter 3 were met in the implementations of the algorithm and how the different implementations methods used for the various transformations of the AES algorithm offered performance benefits.

Chapter 6: Testing

6.1 Introduction

The overall aim of this dissertation was measure was to measure the performance of software implementations of the AES algorithm developed on various platforms. This section presents how the software developed for this project was tested through development phase and how the performance testing of the software on the different platforms was carried. The performance results (see appendix C) of the AES implementations are also discussed in detail.

6.2 Software Testing

The Appendix of the FIPS [7], contains example vectors, including immediate values for the key scheduling for all the input keys (128, 192 and 256 bits), and for the encryption and decryption round transformations of the state array (128 bits block). All these test vectors are in hexadecimal notation, with each pair of digits representing a byte as described in section 2.2.1. Using these test vectors the AES implementations developed for this project were tested using black box testing. The black box testing focused on whether each round transformation or the key scheduling of a specific input key for the Rijndael algorithm produced the same test vectors as the ones in the FIPS [7].

Throughout the implementation phase of the Rijndael software developed for this project, each transformation function implemented was tested using the test vectors in the FIPS to ensure that it produced the right functionality. In the case of the key scheduling, a test vector key was input into the key schedule function, and the resulting output keys producing from running the function were compared to the ones in the FIPS appendix. By testing each of round transformation functions implemented individually, the errors were solved efficiently before the AES implementation became complicated. Tests were also carried out on each of the completed C++ and 68000 assembly language implementations using the same test vectors to ensure that same correct functionality was maintained by the individual round transformation functions and the key schedule function when they were treated as one program.

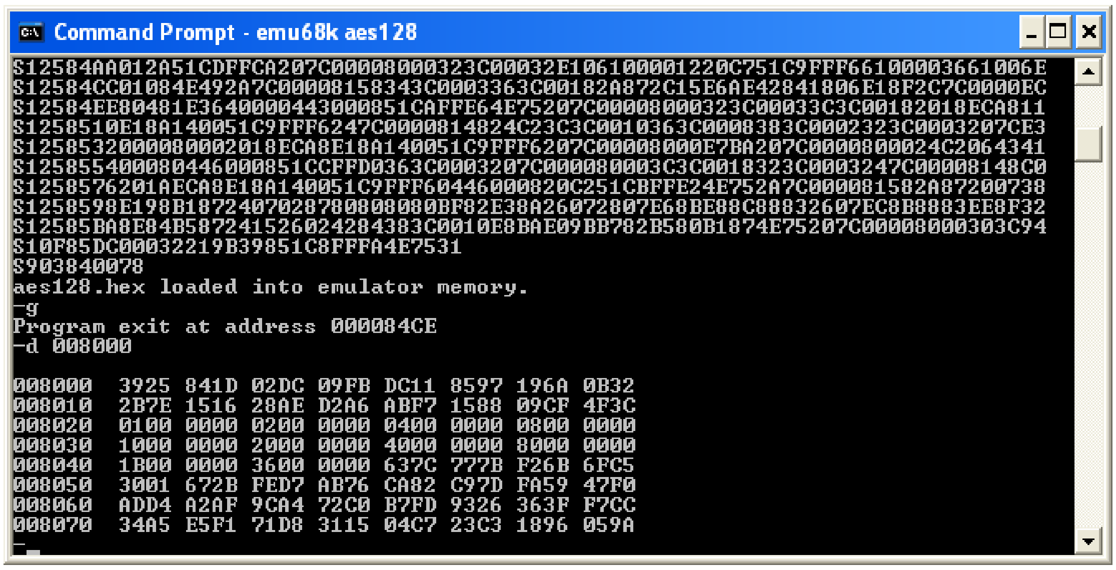
The kind test vectors used for testing the functionality of Rijndael implementations were of the following form:

```
Plaintext: 3243f6a8885a308d313198a2e0370734

Input key: 2b7e151628aed2a6abf7158809cf4f3c

Ciphertext: 3925841d02dc09fbdc118597196a0b32
```

The 32 digit hexadecimal numbers represent the128 bits block that is input into the state array of the AES algorithm, a 128 bits input key and the expected 128 bits output block after encryption. The command prompt window below shows the resulting output after running the above test vectors on the 68000 assembly implementation.

- `-g` is the emulator command that executes the program loaded in its memory. In this case it is the 68000 assembly language implementation for 128 bits input block with a 128 bits input key.

- `-d 080000` is the emulator command that displays its memory contents beginning at address 080000. This is address where the transformed state array is stored in the implementation. By comparing hexadecimal value at this address with the test vector ciphertext hexadecimal value, the author was able to establish whether the implementation was functioning correctly.



```
S12584AA012A51CDFFCA207C00008000323C00032E106100001220C751C9FFF66100003661006E
S12584CC01084E492A7C00008158343C0003363C00182A872C15E6AE42841806E18F2C7C0000EC
S12584EE80481E3640000443000851CAFFE64E75207C00008000323C00033C3C00182018ECA811
S1258510E18A140051C9FFF6247C00008148 24C23C3C0010363C0008383C0002323C0003207CE3
S125853200008000201 8ECA8E18A140051C9FFF6207C00008000E7BA207C0000800024C2064341
S1258554000804460008 51CCFFD0363C0003207C000080003C3C0018323C0003247C000081 48C0
S12585762 01AECA8E18A140051C9FFF6044600082 0C251CBFFE24E752A7C000081582A87200738
S1258598E198B18724070287808080 80BF82E38A26072807E68BE88C88832607EC8B8883EE8F32
S12585BA8E84B5872415260242843 83C0010E8BAE09BB782B580B1874E75207C00008000303C94
S10F85DC00032219B39851C8FFFA4E7531
S903840078
aes128.hex loaded into emulator memory.
-g
Program exit at address 000084CE
-d 008000

008000  3925 841D 02DC 09FB DC11 8597 196A 0B32
008010  2B7E 1516 28AE D2A6 ABF7 1588 09CF 4F3C
008020  0100 0000 0200 0000 0400 0000 0800 0000
008030  1000 0000 2000 0000 4000 0000 8000 0000
008040  1B00 0000 3600 0000 637C 777B F26B 6FC5
008050  3001 672B FED7 AB76 CA82 C97D FA59 47F0
008060  ADD4 A2AF 9CA4 72C0 B7FD 9326 363F F7CC
008070  34A5 E5F1 71D8 3115 04C7 23C3 1896 059A
-
```

The same testing procedure described above was used on the C++ implementation to ensure correct functionality. These functionality tests carried out on the Rijndael implementations during the software development stage were able to conform that software was functioning correctly before the performance testing was carried out.

6.3 Performance Testing

The performance testing of the Rijndael implementations developed for this project needed to be carried out on different platforms as specified in the objectives (see section 1.3.1). The performance tests aimed to determine the execution times (clock cycles) of the variations of the Rijndael implementations on the different platforms. The C++ implementation was run on the following BUSC and personal computers that represent the 32-bit and multi-processor platforms:

- **Mary**: SUN ultra E450 – with four 296MHz Ultra SPARC II processors and 2GBytes of RAM
- **Midge**: SUN ultra E3500 – with eight 400MHz Ultra SPARC II processors and 4GBytes of RAM

- **amos**: SUNFIRE 480R – with four 900MHz Ultra SPARC III processors and 16GBytes of RAM

- **Death**: DELL POWEREDGE 2650 – with four 1.8GHz Intel Xeon processors and 2GBytes of RAM

- **Laptop PC**: ACER – with one 2.6GHz Intel Celeron processor and 256MB DDR SDRAM

The C++ standard library functions `clock()`and `time()` were used to measure the execution times for the Rijndael cipher and inverse cipher implementations on the above platforms. The pseudo code below, illustrates how these time functions were used to obtain the performance results (see appendix C) of the Rijndael C++ implementation.

<u>clock()</u>

```
t0 = clock()
 for (i = 0; i < 100000; i++)
   void aes_encrypt();
t1 = clock()

execution time = ((t1 - t0)/ (CLOCK_PER_CYCLE))/ 100000
```

`clock()`– returns the time in terms of CPU clock cycles, it is therefore divided by `CLOCK_PER_CYCLE` which converts the machine cycles into time in seconds.

<u>time()</u>

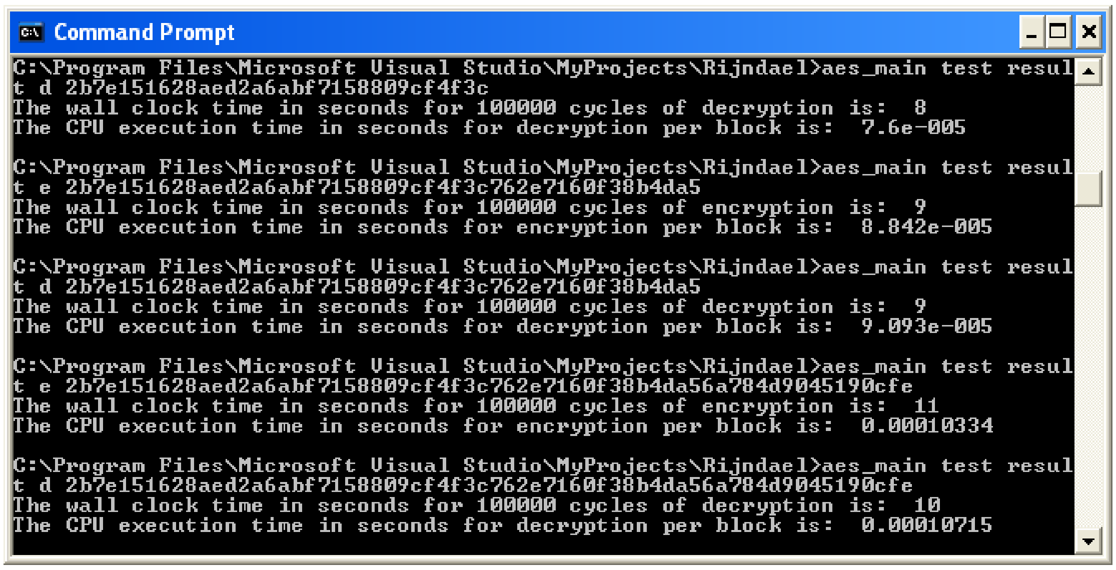
```
start = time()
 for (i = 0; i < 100000; i++)
   void aes_decrypt();
end = time()

execution time = (end - start)
```

`time()`– returns the elapsed time from the moment the repeated iterations of the encrypt or decrypt function starts until it terminates. This is called the wall clock time.

Multiple executions of the encryption and decryption functions were performed to ensure that overall cycle took a reasonable length of time and to allow the results to stabilise. It was decide for this project to use 100,000 iterations for the performance testing. After experimenting with 10,000, 500,000 and 1,000,000 iterations, it was found that 100,000 iterations produced similar and stable result as the 500,000 and 1,000,000 iterations.

The performance testing of the Rijndael C++ implementation was carried by compiling the C++ program and running it on the various platforms. On the UNIX machines Mary, amos and Midge the implementation was compiled using the g++

compiler and on the windows machines Microsoft Visual C++ was used for compiling. The Command prompt window below illustrates how the performance results (see appendix C) were obtained on the windows machines.



```
C:\Program Files\Microsoft Visual Studio\MyProjects\Rijndael>aes_main test resul
t d 2b7e151628aed2a6abf7158809cf4f3c
The wall clock time in seconds for 100000 cycles of decryption is:  8
The CPU execution time in seconds for decryption per block is:  7.6e-005

C:\Program Files\Microsoft Visual Studio\MyProjects\Rijndael>aes_main test resul
t e 2b7e151628aed2a6abf7158809cf4f3c762e7160f38b4da5
The wall clock time in seconds for 100000 cycles of encryption is:  9
The CPU execution time in seconds for encryption per block is:  8.842e-005

C:\Program Files\Microsoft Visual Studio\MyProjects\Rijndael>aes_main test resul
t d 2b7e151628aed2a6abf7158809cf4f3c762e7160f38b4da5
The wall clock time in seconds for 100000 cycles of decryption is:  9
The CPU execution time in seconds for decryption per block is:  9.093e-005

C:\Program Files\Microsoft Visual Studio\MyProjects\Rijndael>aes_main test resul
t e 2b7e151628aed2a6abf7158809cf4f3c762e7160f38b4da56a784d9045190cfe
The wall clock time in seconds for 100000 cycles of encryption is:  11
The CPU execution time in seconds for encryption per block is:  0.00010334

C:\Program Files\Microsoft Visual Studio\MyProjects\Rijndael>aes_main test resul
t d 2b7e151628aed2a6abf7158809cf4f3c762e7160f38b4da56a784d9045190cfe
The wall clock time in seconds for 100000 cycles of decryption is:  10
The CPU execution time in seconds for decryption per block is:  0.00010715
```

For the 68000 assembly language implementation of the AES, the performance testing was carried out by manually calculating the number of clock cycles for the encryption, decryption and key scheduling. The appendix C of "The 68000 microprocessor" [8], contains listings of the 68000 microprocessor instruction execution times in terms of external clock (CLK) periods. Using these instruction execution times, the author was able to calculate the number of clock cycles required to execute the encryption, decryption and key scheduling implementations. Although this method used to determine the performance results of 68000 implementations is accurate, it is laborious and error prone. The performance results of the 68000 implementations can be found in appendix C.

6.4 Performance Analysis

The performance of the AES (Rijndael) or any other algorithm is dependant on many factors. Some of the factors that have an influence on the performance results of the AES are; the programming language and compiler used by the implementation, the processor's speed and the size of its cache, and the amount of memory (RAM) the machine on which the implementation is tested has. This makes it difficult to compare the performance results of the implementations on different platforms, but even with this constraint some consistent trends were observed in the performance results produced by the AES implementations developed for this project (see appendix C).

The encryption, decryption and key scheduling which is similar for both the encryption and decryption for the AES implementations developed for this project, got slower as the size of the input keys increased. Rijndael specifies more rounds for larger key size, therefore affecting the speed of the key expansion, encryption and decryption of its variations. The Rijndael decryption process was slower than the encryption process for all the variations of the implementations, this is mainly due to the fact that the inverse *MixColumns* transformation takes considerable more clock cycles to execute compared to the cipher *MixColumns* transformation. This was best demonstrated by following execution times calculated for the 68000 implementation, where the inverse *MixColumns* transformation required 1500 clock cycles per 32-bits word compared to the 340 cycles per 32-bits word required for the encryption *MixColumns* transformation.

Hand written assembly implementations of the AES generally performs better than compiler optimised assembler. This is confirmed by Gladman's comparison of an x86 assembler (Pentium) code implementation with C++/C implementations on Pentium family processors where the assembler implementation was found to be 20% faster. It was difficult to compare the performance results of the 68000 assembler produced for this project to those of the C++ implementation since they were measured on different processors unlike the Gladman's implementations mentioned before. When the execution times for the 68000 implementations developed for this dissertation were compared to the Motorola 68HC08 assembler performance figures in "The Rijndael block cipher" [3], the 68HC08 implementation results were found to be 10 time faster. This suggest that the 68000 assembly language implementation developed for this project could have been optimised further to produce better results, but this was not possible due to the time constraint on the software development phase for this dissertation.

The execution times for the Rijndael C++ implementation on the windows machines with Intel processors were faster than the ones for UNIX machines with SPARC processors. This could be attributed to various factors, the obvious ones being the different compilers and processors of the machines. The windows machines used Microsoft Visual C++ complier which appears to offer performance benefits compared with the g++ compiler used on the UNIX machines. The difference in processor speeds and cache sizes could also have had an effect on the performance results.

When the execution times of the Rijndael C++ implementation on UNIX machines with SPARC processors are compared, the C++ implementation performs best on *amos*, which has the UNIX machine with the highest processor speed per processor and the most RAM. The number of processor on the machine did not seem to affect the performance of the implementation, but this could be due to fact that implementation was not developed to take advantage of the multiple processors. The performance figures of the Rijndael implementation on the Intel Xeon and Intel Celeron were almost identical; this could be put down to the similarity of the underlying architecture of the two processors or the optimised assembler of the implementation produced by the Microsoft Visual C++ compiler.

The Rijndael C++ implementation developed for this project was not tested on a 64 bit platform as specified in the aims of the dissertation because there was none available at the time the performance testing was carried out. A 64 bit processor though should produce better performance results than the 32 bit processors used for this project, due to its ability to execute two 32 bit instructions at once.

In order to achieve the kind of performance figures Gladman [4] and Lipmaa [10] obtained for their C++ implementations, the Rijndael C++ implementation developed for this project should have been implement using look-up tables (see section 2.5) , but this approach is memory intensive and could not be used on the 8- bit platform due to limited memory. In the case of Gladman's C++ implementation for the highest speed, the code uses a maximum of 20 tables of 256 32-bit words, a total of 20Kbytes of data. Since the aim for this dissertation was to compare the performance of the Rijndael implementations on different platforms, it was decided earlier on in the requirement elicitation phase that the method of implementation for both the 68000 assembly language and C++ had to be the similar so that their performances could be compared.

It is evident from the performance results obtained from the performance testing of the Rijndael implementations developed for this project, that the AES algorithm can be implemented to run at very high speeds on both 8-bit and 32-bit platforms. The 128 bits input key variation would be recommended for the 8-bit platform implementation as it offers better performance and require less memory which is an issue on such platforms. As for the 32 bit platforms and higher where memory is not an issue, the performance of the different input key variations of the AES implementation seem to differ very slightly, so the choice would be down to personal preference.

Chapter 7: Critical evaluation

7.1 Evaluation of Work

The overall aim of this dissertation was to develop a software implementation of the AES (Rijndael) that could be performance tested on an 8-bit platform that is representative of the microprocessors used in Smart cards like Debit/ Credit cards and also develop another implementation of the AES that could be performance tested on 32-bit, 64-bit and multi-processor platforms that are representative of personal computers and High-end servers that are use today. The outcome of this project was the implementation of the AES in the 68000 assembly language to run on the 16-bit 68000 microprocessor and in Standard C++ to run on the 32-bit or higher processors in personal computers and High-end servers. Although the 68000 microprocessor is not an 8-bit processor, it was still representative of the microprocessor the project aimed to performance test the AES on. This project therefore met all its aims and objectives (see section 1.3) with the exception of performance testing the C++ implementation developed on a 64-bit platform, due to lack of access to one.

A considerable amount of time was spent in the early stages of project trying to understand mathematical concepts and the different transformations of the Rijndael algorithm. The requirements for this project were then captured through extensive reading on the AES and examining already existing Rijndael implementations in the public domain. Since computer with 68000microprocesors were not readily available, it was decided that 68000 simulator would be used for this project as specified in the requirements specification (see Chapter 3). Before any software could be developed for the project the author had to learn how to write programs in the 68000 assembly language and learn more about C++ programming. Decisions also had to be made on how best the Rijndael algorithm was going to be implemented, whether or not to use the very efficient lookup table. The reasons why the very efficient table implementation method for AES was not used for the C++ implementation for this project were discussed in section 6.4.

Even though it was decided not to use the table implementation for this project, performance of the final implementations was at the core of every decision made, when deciding which particular method of implementation should be used for the particular round transformations that make up the AES encryption and decryption algorithm. As an incremental process was followed when developing the software for this project, this resulted in stable modular implementations that could easily be tested. The C++ implementation was developed using object oriented techniques provided the programming language, resulting in clear, easy to read and understandable code. The endianness of the processors on which the C++ implementation was going to be performance tested had to taken in to account when writing the program. This was a challenging problem but was solved by writing a function that did the conversion of the input block (16 digit hexadecimal value) into the state array rather than letting the processor do the conversion. This exact function was also implemented successfully for the input keys.

For the finite field multiplication in C++ implementation, inline functions were used which provided performance benefits and also used was the efficient multiplication by repeated shift method (see section 2.1.4) which offered further performance gains. AS for the 68000 assembly language implementation an even more efficient method was used for the implementation of the encryption and decryption *MixColumns* transformations. This performance enhancing method (see section 5.4.3 and 5.4.4) took advantage fact that, despite the 68000 microprocessor being a 16-bit processor it provides instructions that operate on 32 bit words. This capability offered by the 68000 microprocessor was utilized through the rest of the assembly language implementation of the AES algorithm for this project. A considerable amount of time for the project was spent on implementing the AES algorithm as it also involved testing each implemented function of the algorithm as soon it was developed. This approach to software development applied to this project helped avoid the challenges and pitfalls of trying to debug already complicated code. The result was thoroughly tested, easily manageable and fully functioning implementations of the AES algorithm.

The performance testing the AES implementations offered the biggest challenge of the project. Obtaining accurate and repeatable time proved more difficult than I expected especially for the C++ implementation that was tested on various platforms. Although the performance testing for the 68000 implementations of the AES for this project was done manually, it produced the more accurate results since they were calculated from the instruction execution times of the 68000 microprocessor. The performance testing for the C++ implementation used the `clock()` and `time()` functions from C++ standard library to measure the execution times for decryption, encryption and key scheduling subroutines (see section 6.3). This method though produced some consistent performance results across the different platforms is not the most reliable and was also flawed in that, it used the same input key for the repeated cycles leading to optimization. Gladman's performance figures were obtained using the more reliable RDTSC (Read Time Stamp Counter) function, used to measure clock cycles on Intel Pentium processors and also using a random number generator to produce different set of input keys for each of the iterations, therefore preventing the optimisation for specific set of keys. Due to the time constraint on this project and the requirement to have a consistent time measuring method for the C++ implementation on the different platforms, it was decided that since the RDTSC function only measures the clock cycles on Pentium processors, this method could not be used for this project.

7.2 Further work

The performance testing methods used for used this project could definitely have been improved, but with there not being a general consensus as to what are the best testing methods to use when measuring the performance of an AES implementation, further research needs to be done in this area and this though was beyond the scope of this project.

Finally since this project only considered the FIPS standard Rijndael algorithm with a fixed input block of 128 bits, further work could be done on the performance of the other variations of Rijndael algorithm with variable input block that are not included in the standard.

7.3 Concluding Remark

Overall it was felt that the project was a success. The Advanced Encryption Standard (Rijndael) was implemented in 68000 assembly language and standard C++, and performance tested on various platforms. Although performance results obtained from the implementations developed for this project are not the best available, much was learned about the implementation and performance of the AES (Rijndael), C++ and 68000 assembly language programming in a short space of time.

Bibliography

[1] J. Daemen, V. Rijmen, “The Design of Rijndael,” Springer-Verlag, Berlin, 2002.

[2] B. Beckett, “Introduction to Cryptology,” Blackwell Scientific, Oxford, 1988.

[3] J. Daemen, V. Rijmen, “The block cipher Rijndael,” available from NIST’s AES homepage, URL: **http://www.nist.gov/aes**.

[4] B. Gladman, “Implementation of AES (Rijndael) in C/C++ and Assembler,” URL: **http://fp.gladman.plus.com/crytography_technology/rijndael**.

[5] Wikipedia, “The Free Encyclopedia,” URL: **http://en.wikipedia.org/wiki**.

[6] J. Daemen, V. Rijmen, B. Gladman, “A Specification for Rijndael, the AES Algorithm,” URL: **http://www.nist.gov/aes**.

[7] Federal Information Processing Standards Publications 197, November 2001, available from NIST’s AES homepage, URL: **http://www.nist.gov/aes**.

[8] J. Antonakos, “The 68000 Microprocessor: hardware and software principles and applications,” Prentice-Hall, 1996

[9] H. Schildt, “C++ from the GROUND UP,” McGraw-Hill/Osborne, 2003

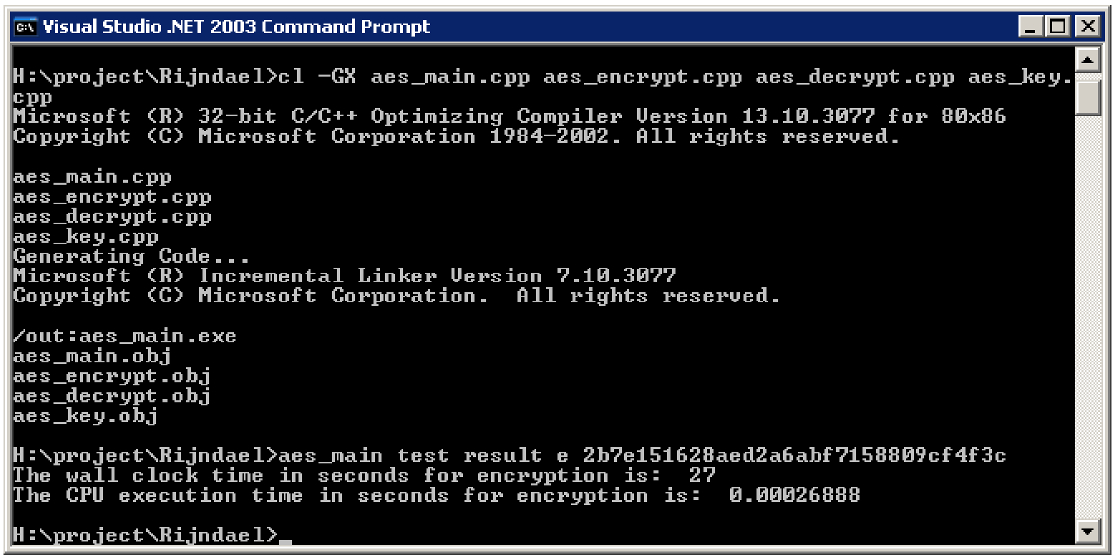
[10] K. Aoki, H. Lipmaa, “Fast implementation of AES candidates” in *submitted for publication – Third AES Candidate Conference, New York City, USA*, 2001

Appendix A: Rijndael C++ Code

This section contains the C++ implementation of the AES (Rijndael) developed for developed for this project. Below are the files contained in the Microsoft Visual C++ Project file, `Rijndael.dsw` in which they AES C++ implementation were developed.

- **`aes.h`**
- **`aes_encrypt.cpp`**
- **`aes_decrypt.cpp`**
- **`aes_key.cpp`**
- **`aes_main.cpp`**
- **`result`**
- **`test`**

This code can be compiled and run in the using the Microsoft Visual C++ compiler illustrated in the command prompt as shown below.



```
H:\project\Rijndael>cl -GX aes_main.cpp aes_encrypt.cpp aes_decrypt.cpp aes_key.
cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

aes_main.cpp
aes_encrypt.cpp
aes_decrypt.cpp
aes_key.cpp
Generating Code...
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:aes_main.exe
aes_main.obj
aes_encrypt.obj
aes_decrypt.obj
aes_key.obj

H:\project\Rijndael>aes_main test result e 2b7e151628aed2a6abf7158809cf4f3c
The wall clock time in seconds for encryption is:  27
The CPU execution time in seconds for encryption is:  0.00026888

H:\project\Rijndael>_
```

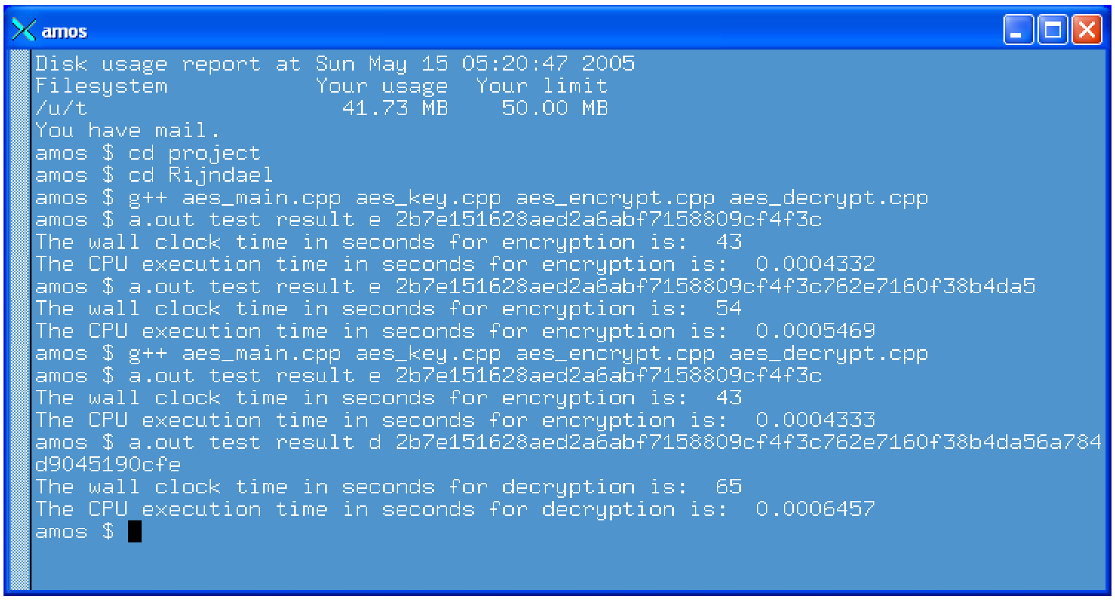
To compile this code using the Microsoft Visual C++ compiler use the following command below:

```
Cl -GX aes_main.cpp aes_key.cpp aes_encrypt.cpp aes_decrypt.cpp
```

To compile this code using the UNIX g++ compiler use the following command below:

```
g++ aes_main.cpp aes_key.cpp aes_encrypt.cpp aes_decrypt.cpp
```

The compiled C++ code can then be run in the UNIX shell or the windows command prompt as illustrated in the UNIX shell using the following commands for the different input key:



```
Disk usage report at Sun May 15 05:20:47 2005
Filesystem            Your usage  Your limit
/u/t                    41.73 MB    50.00 MB
You have mail.
amos $ cd project
amos $ cd Rijndael
amos $ g++ aes_main.cpp aes_key.cpp aes_encrypt.cpp aes_decrypt.cpp
amos $ a.out test result e 2b7e151628aed2a6abf7158809cf4f3c
The wall clock time in seconds for encryption is:  43
The CPU execution time in seconds for encryption is:  0.0004332
amos $ a.out test result e 2b7e151628aed2a6abf7158809cf4f3c762e7160f38b4da5
The wall clock time in seconds for encryption is:  54
The CPU execution time in seconds for encryption is:  0.0005469
amos $ g++ aes_main.cpp aes_key.cpp aes_encrypt.cpp aes_decrypt.cpp
amos $ a.out test result e 2b7e151628aed2a6abf7158809cf4f3c
The wall clock time in seconds for encryption is:  43
The CPU execution time in seconds for encryption is:  0.0004333
amos $ a.out test result d 2b7e151628aed2a6abf7158809cf4f3c762e7160f38b4da56a784
d9045190cfe
The wall clock time in seconds for decryption is:  65
The CPU execution time in seconds for decryption is:  0.0006457
amos $ █
```

<u>`128 input key`</u>

```
aes_main test result [d/e] 2b7e151628aed2a6abf7158809cf4f3c
```

<u>`192 input key`</u>

```
aes_main test result [d/e] 2b7e151628aed2a6abf7158809cf4f3c762
e7160f38b4da5
```

<u>`128 input key`</u>

```
aes_main test result [d/e] 2b7e151628aed2a6abf7158809cf4f3c762
e7160f38b4da56a784d9045190cfe
```

The input block is input through the **`test`** file and the output block is written to the **`result`** file.

`[d/e]`- this is the encryption or decryption option.

Appendix B: Rijndael 68000 Assembler Code

This section contains the AES (Rijndael) implementations written for this project in 68000 assembler language. Below are the assembler files that contain code of the different implementations.

- **`aes128`**
- **aes192**
- **aes256**
- **Inv128**
- **Inv192**
- **Inv256**

For this project the above 68000 implementations of the AES were assembled and run on the Antonokos' 68000 simulator using the following commands. You need to install the following software on your computer before using the commands below to run this 68000 assembly code **asm68k (assembler)** and **emu68k (emulator).**

To assemble the code use the following command;

```
asm68k aes128.asm
```

When the above code is executed the assembler creates three files with extension **.hex, .list** and .**obj**. Then to execute the above assembled program use the following command.

```
emu68k aes128.hex
```

See section 6.2 for an example how the code is run in the DOS command window.

Appendix C: Test Results

This section contains the performance results obtained from performance testing (see section 6.3) the AES implementations developed this project on different platforms.

68000 assembly language implementation

These execution times were calculated manually using the instruction execution times for a 68000 microprocessor. For example on the 8MHz 68000 microprocessor each clock cycle executes in 125 nanoseconds. Therefore the execution time in for encryption of 128 bits input block with a 128 bits input key would be equal to (125 x 72991) / 1 x 10^{-9} = 0.009124 seconds.

Table C.1: Execution times in clock cycles for the Rijndael in 68000 assembler

key length	block length	Encryption	Decryption	key Scheduling
128	128	72991	97492	15940
192	128	87863	117808	16008
256	128	102735	138120	20302

C++ implementation

For this implementation the C++ standard library time functions `clock()` and `time()` were used to measure the execution times of the AES implementation as described in section 6.3.

Midge: SUN ultra E3500 – with eight 400MHz Ultra SPARC II processors and 4GBytes of RAM

Table C.2: Execution time in seconds for Rijndael in C++ on Midge

key length	block length	Encryption CPU time	Encryption wall clock time	Decryption CPU time	Decryption wall clock time
128	128	0.0007441	0.00074	0.0008591	0.00086
192	128	0.0009385	0.00094	0.0010767	0.00108
256	128	0.0010929	0.00109	0.001278	0.00128

Mary: SUN ultra E450 – with four 296MHz Ultra SPARC II processors and 2GBytes of RAM

Table C.3: Execution time in seconds for Rijndael in C++ on Mary

key length	block length	Encryption CPU time	Encryption wall clock time	Decryption CPU time	Decryption wall clock time
128	128	0.0009978	0.00100	0.0011834	0.00118
192	128	0.0012524	0.00125	0.0014475	0.00145
256	128	0.0014637	0.00147	0.0016581	0.00167

amos: SUNFIRE 480R – with four 900MHz Ultra SPARC III processors and 16GBytes of RAM

Table C.4: Execution time in seconds for Rijndael in C++ on amos

key length	**block length**	**Encryption CPU time**	**Encryption wall clock time**	**Decryption CPU time**	**Decryption wall clock time**
128	128	0.0004323	0.00043	0.0004551	0.00045
192	128	0.0005457	0.00055	0.0005468	0.00055
256	128	0.0006365	0.00063	0.0006434	0.00065

Laptop PC: ACER – with one 2.6GHz Intel Celeron processor and 256MB DDR SDRAM

Table C.5: Execution time in seconds for Rijndael in C++ on Intel Celeron

key length	**block length**	**Encryption CPU time**	**Encryption wall clock time**	**Decryption CPU time**	**Decryption wall clock time**
128	128	0.0000707	0.00007	0.0000760	0.00008
192	128	0.0000884	0.00009	0.0000909	0.00009
256	128	0.0001033	0.00011	0.0001072	0.00010

Death: DELL POWEREDGE 2650 – with four 1.8GHz Intel Xeon processors and 2GBytes of RAM

Table C.6: Execution time in seconds for Rijndael in C++ on Intel Xeon

key length	**block length**	**Encryption CPU time**	**Encryption wall clock time**	**Decryption CPU time**	**Decryption wall clock time**
128	128	0.0000802	0.00008	0.0000806	0.00008
192	128	0.0000905	0.00009	0.0000903	0.00009
256	128	0.0001059	0.00011	0.0001145	0.00012